Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Plataforma Unificada e Extensível para Colaboração com Dispositivos Heterogéneos

Bernardo Luís Silva Ferreira (aluno nº 28300)

2º Semestre de 2009/2010
28 de Julho de 2010

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Plataforma Unificada e Extensível para Colaboração com Dispositivos Heterogéneos

Bernardo Luís Silva Ferreira (aluno nº 28300)

Orientadores
Prof. Doutor Henrique João Lopes Domingos
Engenheiro Paulo Chainho

Trabalho apresentado no âmbito do Mestrado em Engenharia Informática, como requisito parcial para obtenção do grau de Mestre em Engenharia Informática

2º Semestre de 2009/2010
28 de Julho de 2010

Agradecimentos

Desejo expressar o meu agradecimento a todos os que tornaram possível esta dissertação:

Em primeiro lugar, ao Engenheiro Paulo Chainho e ao Professor Henrique João Domingos, por me terem proposto para esta dissertação e pelas recomendações e orientação durante o percurso do trabalho desenvolvido.

De seguida, a toda equipa do Pólo de Lisboa da PT Inovação, onde fui integrado no contexto deste trabalho, incluindo o Mauro Batista, Toni Barata, Neutel Rodrigues, Pedro Gomes, Pedro Taborda, Pedro Correia, Hugo Fernandes e João Sousa, pelos conselhos, apoio e presença durante a realização desta dissertação.

Gostaria de agradecer também a todos os meus amigos e colegas da FCT/UNL, em especial ao Fábio Neves, Miguel Domingues, Jorge Costa, Nuno Parreira, Pedro Nunes, Tiago Araújo, Ivo Rodrigues, Tiago Santos e Sérgio Pereira, pelos seus apoios e auxílios ao longo de todo o decorrer do mestrado que é agora terminado com esta dissertação.

Por fim, gostaria de agradecer a toda a minha família, que sem o seu suporte nada deste trabalho poderia ter sido realizado.

Resumo

Nos últimos tempos têm surgido vários serviços de cariz social centrados nos utilizadores, propiciando diferentes formas de interacção, convívio e colaboração. Um exemplo destes é o Google Wave. Ao juntar mecanismos de controlo de concorrência optimística com diversas funcionalidades colaborativas, o Google Wave constitui um serviço colaborativo em tempo real que funde ferramentas de colaboração síncronas e assíncronas.

Dentro do contexto da evolução das comunicações colaborativas e seguindo estas novas tendências, a PT Inovação está a desenvolver uma Plataforma Colaborativa designada por PUC (Plataforma Unificada de Colaboração), cujo objectivo é simplificar o processo de construção de software colaborativo para dispositivos heterogéneos, ao juntar os mais diversos tipos de recursos de colaboração sobre uma única interface colaborativa genérica que pode adoptar diferentes redes de comunicação.

Esta dissertação enquadra-se no desenvolvimento da plataforma PUC, tendo como objectivo expandi-la com funcionalidades e princípios apresentados pelo Google Wave. As contribuições desenvolvidas na dissertação partiram do estudo dos conceitos e tecnologias do Google Wave, tendo em vista (1) integrar uma camada de colaboração com as funcionalidades do Goggle Wave no PUC, (2) conceber, normalizar e desenvolver uma interface portável para suporte de aplicações heterogéneas no Google Wave, (3) expandir o modelo de dados do PUC e reestruturar o seu sistema de informação interno, suportando o modelo de excertos de frases, (4) adição de uma camada para disseminação assíncrona de eventos para integração uniforme e transparente das aplicações, (5) suportar funcionalidade de repositório de dados de gravação e reprodução de medias heterogéneos, (6) conceber e suportar uma camada de extensibilidade baseada em *widgets* colaborativas externas para reutilização normalizada dos serviços *middleware* oferecidos pela plataforma, sem modificação da sua estrutura interna.

No final, a plataforma de colaboração desenvolvida, baseada na plataforma PUC e estendida com as funcionalidades e conceitos do Goggle Wave, permitirá criar aplicações colaborativas complexas, de características multi-síncronas, que ofereçam experiências colaborativas ainda mais ricas aos utilizadores.

Palavras-chave: Plataforma Unificada de Colaboração, Edição Colaborativa de Documentos, Dispositivos Heterogéneos, Extensibilidade, Consistência de Dados, Transformação de Operações, Framework Colaborativa, Colaboração Multi-Síncrona.

Abstract

In the last years multiple social services centered in the users have arisen, providing different forms of interaction, socialization and collaboration. An example is Google Wave. By joining together optimistic concurrency control mechanisms with diverse collaborative features, Google Wave offers a real time collaborative service that merges synchronous and asynchronous collaborative tools.

Within the context of the evolution of collaborative communications and following these new tendencies, PT Inovação is developing a Collaborative Framework designated by PUC (Unified Collaborative Platform), whose objective is to simplify the construction process of collaborative software for heterogeneous devices, by joining the most diverse kinds of collaborative resources under a unique generic collaborative interface that can adopt different communication networks.

This dissertation is framed in the development of PUC, aiming to expand it with the features and principles presented by Google Wave. The contributions developed in the dissertation came from the study of the concepts and technologies of Google Wave, having in sight (1) the integration of a collaborative layer with Google Wave's features in PUC, (2) the conception, normalization and development of a portable interface in order to support heterogeneous applications in Google Wave, (3) the expansion of PUC's data model and the restructure of its internal information system, supporting the model of utterances, (4) addition of a layer for asynchronous dissemination of events for uniform and transparent integration of the applications, (5) the support of data repository feature for recording and playback of heterogeneous media, (6) the conception and support of an extensibility layer based on collaborative external widgets for reutilization in a standardized way of the middleware services offered by the platform, without modification of its internal structure.

In the end, the collaboration platform developed, based on the platform PUC and extended with Google Wave's features, will allow to create complex collaborative applications, with multi-synchronous features, that offer even richer collaborative experiences to its users.

Keywords: Unified Collaborative Platform, Collaborative Document Edition, Heterogeneous Devices, Extensibility, Data Consistency, Operational Transformation, Collaborative Framework, Multi-Synchronous Collaboration.

Conteúdo

Lista de Figuras

1. Introdução

1.1. Motivação

Com o aparecimento da Web 2.0, têm surgido uma panóplia de novos serviços e aplicações que vêem a Web como uma plataforma. Estas aplicações procuram oferecer aos seus utilizadores uma experiência extremamente rica, com conteúdos cada vez mais dinâmicos e uma visão do software como serviço centrado no utilizador. Por outro lado, os utilizadores estão cada vez mais ligados às novas tecnologias e procuram utilizar todo o tipo de médias disponíveis. A tendência é que estas sejam utilizadas de forma contínua e sem rupturas no dia-a-dia e a troca entre as várias seja cada vez mais natural e inconsciente. Como *Mark Weiser* uma vez disse: "The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it" [7]. Traduzindo, fica algo com um sentido parecido com: as tecnologias mais importantes são as que não se vêem, que se fundem com o dia-a-dia de cada um de tal forma que o seu uso seja inconsciente. A tendência das aplicações nesta era da Web 2.0 é exactamente esta, tornarem-se cada vez mais indistinguíveis da vida quotidiana.

A liderar a era da Web 2.0 e o lançamento de novas tecnologias deste tipo, encontramos a Google. Desde a sua criação a Google tem sempre procurado revolucionar o mundo tecnológico e a forma como as pessoas interagem com este. Do Google Maps ao Google Docs [17], o contributo da Google para o crescimento da Web 2.0 e deste tipo de tecnologias tem sido notável. Neste contexto, a Google lançou recentemente um novo produto denominado de Google Wave [5]. Este produto pretende revolucionar a maneira como as pessoas comunicam entre si, permitindo que grupos de pessoas discutam e editem documentos e conversas de forma colaborativa e em tempo quase real. O Wave, em fase de *preview* na altura de escrita deste documento, foi lançado como um projecto open-source de forma a poder contar com o contributo de toda a comunidade Web. Neste sentido, para além de terem sido disponibilizados os protocolos usados, versões de protótipo de servidores e simples clientes, foram também disponibilizadas APIs que permitem estender e enriquecer o Wave com as mais variadas funcionalidades. A visão dos criadores do Wave é a seguinte: “Como seria o E-Mail se fosse inventado hoje?” [5]. Esta visão é um pouco irrealista, pois a Google não está realmente a reinventar o E-Mail mas sim a explorar uma nova tendência nas comunicações virtuais. No entanto, o Google Wave não deixa de ter grandes possibilidades e muito potencial para ser explorado.

Como tal, a grande motivação desta dissertação é explorar, estudar e discutir os conceitos e tecnologias introduzidos pelo Google Wave e apresentar um caso de estudo de integração destes numa Plataforma de Colaboração existente. A plataforma referida será brevemente apresentada na próxima secção, juntamente com uma descrição mais detalhada do problema.

1.2. Descrição do Problema

Se abstrairmo-nos um pouco da visão de software como serviço e da Web 2.0, apresentadas na secção anterior, pode-se observar que acima de tudo o Google Wave [5] é um software de Computer-Supported Cooperative Work (CSCW). Uma vasta discussão sobre o tema pode ser encontrada em [2], segundo o qual software de CSCW geralmente divide-se em duas categorias, de acordo com a sua natureza temporal. A primeira categoria é a de software colaborativo assíncrono. Este tipo de software caracteriza-se por ajudar os seus utilizadores a comunicarem entre si mas de forma assíncrona, ou seja, sem ser em tempo real. Um utilizador deixa uma mensagem para outro num dado momento e mais tarde o outro consulta-a. Exemplos são aplicações de E-Mail, Blogs, Fórums, *Workflows* e *Organizational Memory*. A segunda categoria é de software colaborativo síncrono, ou seja, software que assiste um grupo de utilizadores a trabalharem em conjunto em tempo real. Exemplos são aplicações de *WebConferencing*, *DesktopSharing*, *Instant Messaging* e *Media Spaces*.

Antes de continuar convém definir claramente o conceito de comunicação em *tempo real*. Dadas as capacidades das ligações de rede actuais e dadas as distâncias físicas entre os utilizadores habituais de software colaborativo, é impossível oferecer comunicações em tempo verdadeiramente real. No entanto esse conceito pode ser emulado o melhor possível, ficando no melhor caso apenas limitado pelo tempo de trânsito da rede. Este tipo de comunicações chama-se comunicação em tempo quase real. Comunicações que não tenham esta preocupação de serem o mais rápido possíveis mas que conseguem ainda assim transmitir uma sensação de sincronismo e presença dos seus comunicadores, são denominadas de comunicações em tempo semi-real.

Continuando a discussão anterior, dada a distinção temporal entre diferentes tipos de software de CSCW, o Wave apresenta-se como uma mistura dos dois conceitos. Por um lado é síncrono porque os seus utilizadores podem comunicar em tempo quase real, e o Wave pode inclusive ser estendido de forma a incluir serviços de *webconferencing*, entre outros, através das suas APIs. Por outro lado é assíncrono porque as conversas são guardadas persistentemente, podem ser deixadas mensagens sem que os outros utilizadores estejam presentes e um novo utilizador pode ser adicionado em qualquer altura e ter acesso a tudo o que foi feito antes da sua chegada. Este novo paradigma de software colaborativo chamar-se-á de software multi-síncrono.

Esta dissertação foi realizada em ambiente empresarial, mais especificamente em colaboração com a empresa PT Inovação [1]. A PT Inovação, seguindo as tendências descritas na secção anterior, está a desenvolver uma Plataforma Unificada de Colaboração (PUC). Esta

plataforma, que pode também ser vista como uma Framework pois disponibiliza um conjunto de ferramentas para outras aplicações serem desenvolvidas por cima, tem como objectivo integrar as mais recentes tecnologias e conceitos na área da comunicação, juntando *VoIP*, *WebConferencing*, telepresença, *IM*, partilha de aplicações, *whiteboarding*, etc.

Com esta dissertação pretendeu-se desenvolver uma plataforma mais enriquecida e avançada a partir da integração dos conceitos e tecnologias do Wave na plataforma PUC. Pelas suas características, o PUC é essencialmente um software de CSCW síncrono. Ao introduzir o Wave no PUC, a plataforma passará também a ser um software multi-síncrono: por um lado porque possui fortes componentes de colaboração síncrona; por outro, porque o Wave é um forte componente de comunicação assíncrona e só por si também tem mecanismos de comunicação síncrona.

1.3. Objectivo

O objectivo desta dissertação foi o desenvolvimento de uma plataforma unificada e extensível para colaboração entre dispositivos heterogéneos. Com este objectivo em mente partiu-se da Plataforma Unificada de Colaboração da PT Inovação e dos conceitos e tecnologias do Google Wave. Nomeadamente, o desenvolvimento da plataforma passou pela integração do Wave na plataforma PUC, adicionando uma nova funcionalidade colaborativa de edição de documentos ricos e reforçando a unificação de diferentes recursos pela plataforma. Foram também estendidas as funcionalidades da plataforma através da concepção e desenvolvimento de funcionalidades para *Recording* e *Play* dos seus recursos colaborativos. De forma a aumentar a extensibilidade da plataforma base (PUC) foi concebido um modelo de extensibilidade baseado em *widgets* colaborativas. Por fim, de modo a aumentar o suporte a dispositivos heterogéneos na plataforma, a concretização do objectivo desta dissertação passou também pelo desenvolvimento de uma interface portável para o Google Wave.

No final desta dissertação espera-se dar suporte a aplicações colaborativas complexas que possam ser executadas em diferentes dispositivos heterogéneos e que forneçam diversos serviços colaborativos a partir de uma interface unificada e genérica. A figura 1 apresenta um exemplo de uma destas aplicações. A aplicação, a executar num *Smartphone* com o sistema Android 2.1, é composta por um conjunto de *widgets* para as diferentes funcionalidades da plataforma. Na figura é visível o *widget* Wave, usado para interacção directa com o servidor Wave integrado na plataforma. O *widget* usa a interface portável concebida para poder comunicar com o servidor Wave. É também possível ver a *widget* com a lista de participantes da sessão colaborativa. Esta *widget* obtém os seus dados da plataforma, que fica responsável por manter os mesmos sincronizados com o servidor Wave e outros servidores de recursos usados.

Figura 1 - Aplicação cliente da plataforma com widget Wave aberta

1.4. Principais Contribuições da Dissertação

As principais contribuições deste trabalho incidiram, por um lado, sobre o estudo aprofundado de um novo standard que se encontra em ascensão e que acha-se vir ser de grande importância no futuro das comunicações colaborativas, o Google Wave. Por outro as contribuições incidiram sobre o desenvolvimento de uma nova plataforma colaborativa, a partir da plataforma PUC e dos conceitos e tecnologias Wave. Esta plataforma apresenta-se como sendo ainda mais extensível e mais enriquecida do ponto de vista das funcionalidades colaborativas que oferece. A plataforma desenvolvida reforça também o suporte a heterogeneidade através da criação de uma interface portável para o Wave, que será usada pelas suas aplicações cliente.

Portanto, o grande objectivo da dissertação foi integrar o Wave na plataforma PUC, tendo como principal motivação expandir as suas funcionalidades e recursos, extensibilidade e suporte a heterogeneidade. A realização deste objectivo visou dar solução ao desejo de criar uma plataforma colaborativa evoluída que oferecesse experiências multimédia ricas, interligando diversos tipos de recursos. A avaliação dos objectivos passou pela realização de diversos tipos de testes, incluindo testes de performance, de portabilidade e usabilidade, assim como pela concepção e desenvolvimento de um protótipo para controlo de qualidade e consolidação da plataforma desenvolvida. A realização do anterior objectivo envolveu o desenvolvimento das seguintes contribuições específicas, nomeadamente:

- Concepção e desenvolvimento de uma interface portável para o Google Wave, de modo a que variadas aplicações residentes em dispositivos heterogéneos possam comunicar com o servidor Wave;
- Integração do Google Wave na plataforma PUC e adição de funcionalidades de edição colaborativa de documentos de texto rico na plataforma;
- Expansão do modelo de dados da plataforma com a adição de uma entidade para representar o excerto de uma sessão;
- Concepção e desenvolvimento de mecanismos de importação e exportação de documentos para e a partir do Wave, respectivamente;
- Desenvolvimento de um sistema assíncrono para entrega de eventos às aplicações cliente da plataforma;
- Expansão de funcionalidades da plataforma PUC, com a adição de funcionalidades de *Recording* e *Play* baseadas na funcionalidade de reprodução de histórico do Google Wave e nas três contribuições anteriores;
- Introdução na plataforma PUC de um modelo de extensibilidade, baseado em *widgets* colaborativos e na Gadget API do Google Wave.

1.5. Estrutura do Documento

O restante documento estará dividido em mais cinco capítulos: Trabalho Relacionado (capítulo dois), Concepção (capítulo três), Implementação (capítulo quatro), Validação (capítulo cinco) e Conclusões (capítulo seis).

No capítulo do trabalho relacionado será feito um estudo do estado da arte de diferentes tecnologias que de certa forma estão associadas a este trabalho, assim como serão discutidos outros sistemas com o mesmo objectivo que este. Nomeadamente, o capítulo está dividido em quatro grandes secções: Tecnologias para Colaboração; Protocolos e Plataformas para Colaboração; Serviços Colaborativos; Comunicações Bidireccionais em Ambientes Web.

O capítulo da concepção irá demonstrar de um modo formal e abstracto como foram concebidas as soluções para as contribuições desenvolvidas e a arquitectura das mesmas. O capítulo está dividido pelas grandes contribuições desenvolvidas: Interface Portável para o Wave, Integração do Wave na Plataforma, Funcionalidades de *Recording* e *Play* e Modelo de Extensibilidade. O capítulo da implementação mostrará em mais pormenor como foram implementadas as soluções concebidas no capítulo anterior, com especial ênfase nos detalhes técnicos e tecnologias usadas. Este capítulo está organizado como o anterior.

O capítulo da validação apresentará como foram validadas as contribuições desenvolvidas e que conclusões foram retiradas dos resultados obtidos. O capítulo está dividido por família de

testes desenvolvida, incluindo um subcapítulo sobre os testes de portabilidade efectuados, outro sobre testes de performance e um último sobre os testes de usabilidade levados a cabo. O último capítulo apresentará as conclusões obtidas do trabalho desenvolvido, questões que ficaram em aberto no fim da dissertação e as ideias quanto ao trabalho futuro.

2. Trabalho relacionado

Este capítulo pretende explorar melhor o contexto no qual esta dissertação foi realizada e indicar o espectro das soluções existentes hoje em dia. Mais especificamente, serão analisados e discutidos os vários componentes que se acha ser chave no contexto de software colaborativo e das contribuições desenvolvidas: Tecnologias para Colaboração, Protocolos e Plataformas Colaborativas, Serviços Colaborativos e Comunicações Bidireccionais em Ambientes Web.

Na secção sobre tecnologias para colaboração será explorada a temática de controlo de concorrência, mais explicitamente uma técnica optimística conhecida como Operational Transformation.

Na secção dos protocolos e plataformas colaborativas serão analisados os protocolos normalmente usados nas comunicações de software colaborativo e será estudada a plataforma colaborativa da PT Inovação.

Na secção dos serviços colaborativos serão discutidos exemplos de existentes serviços, incluindo o Google Wave, e que tecnologias usam para implementar as suas funcionalidades.

Na secção das comunicações bidireccionais em ambientes Web será feita uma discussão sobre as actuais soluções para este tipo de comunicações.

É de notar que todos os sistemas ou tecnologias estudados neste capítulo relacionam-se de alguma forma com o tema desta dissertação e com o trabalho realizado.

2.1. Tecnologias para Colaboração - Operational Transformation

Operational transformation (OT) é uma tecnologia para controlo de concorrência e preservação de consistência, que tem sido continuamente desenvolvida ao longo das últimas duas décadas, por várias equipas de investigação à escala global. Esta tecnologia, desenvolvida no contexto de edição de documentos em grupo, tem evoluído para também permitir group undo, partilha de aplicações, entre outras.

OT caracteriza-se por ser não bloqueante, ao contrário de mecanismos baseados em locks, usando replicação para conseguir edição de documentos de forma colaborativa em tempo real. Porém, em vez de tentar manter a consistência em todas as réplicas aplicando operações localmente e depois distribuindo o resultado obtido, como é costume noutras soluções não bloqueantes, OT partilha as próprias operações entre réplicas. Estas, ao receber operações a efectuar vindas de outro site, transformam-nas contra a sua réplica local, usando uma função de transformação, e então aplicam a operação transformada.

OT tem evoluído consideravelmente e várias versões diferentes têm sido propostas ao longo do tempo, cada uma cobrindo os problemas e limitações da versão anterior, ou expondo uma aproximação diferente. A primeira implementação de OT apareceu em 1989 no contexto do sistema GROVE [9], e era completamente distribuída. No entanto, o GROVE sofria de problemas de convergência em certas situações específicas e outros sistemas foram então propostos, destacando-se o REDUCE [10], adOPTed [11] e GOT [12], todos distribuídos. Outra aproximação proposta foi a do sistema JUPITER [13], que em vez de ser distribuída usava um servidor central para gerir a consistência em todos os participantes.

Mais à frente neste artigo serão mostrados em detalhe uma aproximação completamente distribuída e uma centralizada, GOTO [14] e JUPITER, respectivamente, seguidos da solução usada pelo Google Wave [15]. No entanto, antes de entrar em detalhes específicos de implementações, é necessário discutir que modelos de consistência e outras propriedades são geralmente usados/garantidos nestas aplicações, assim como a arquitectura geralmente usada.

2.1.1. Modelos de Consistência

Vários modelos de consistência têm sido propostos na comunidade científica, uns no contexto geral de sistemas de edição colaborativa de documentos e outros especificamente para algoritmos de OT. Seguindo o âmbito do presente trabalho, são discutidos os modelos CC e CCI.

- **CC**

Este modelo foi proposto em [9], tendo as siglas CC o significado Convergência e Causalidade. Convergência neste modelo significa que as cópias de um documento partilhado são iguais em todos os sites em quiescência, ou seja, todas as operações geradas foram executadas em todas réplicas. Causalidade significa que se uma operação precede outra causalmente, então deve ser executada antes da outra em todas as réplicas. A relação de causalidade aqui discutida é definida formalmente como a relação *happened-before* de Lamport. O modelo foi proposto em consequência da identificação de dois problemas: divergência e violação de causalidade. No entanto, em [10] foi identificado mais um problema de consistência, separado dos anteriores. Isto levou à proposta do modelo CCI.

- **CCI**

Este novo modelo é idêntico ao anterior, mas adiciona uma nova propriedade: preservação de Intenção. Esta propriedade significa que o resultado de efectuar uma operação num documento deve ser sempre o intencionado, esteja o documento em que estado estiver. A intenção de uma operação é definida como o efeito de execução da mesma no estado em que foi gerada. O problema de preservação de intenção difere essencialmente do problema de convergência porque o último pode sempre ser solucionado usando serialização, ao contrário do primeiro.

2.1.2. Arquitectura

Geralmente, os algoritmos de OT [9, 10, 11, 13] seguem uma arquitectura que divide-se em duas camadas: Uma camada superior com o algoritmo de controlo de transformações e uma camada inferior com as funções de transformação em si.

O algoritmo de controlo de transformações é responsável por saber que operações devem ser transformadas contra uma nova operação que já esteja causalmente pronta, e qual a ordenação das transformações. As funções de transformação transformam uma operação contra outra. Normalmente são usados dois tipos de transformações: transformações inclusivas (IT), em que uma operação é transformada contra outra de forma a que o efeito da segunda seja incluído na transformação; e transformações exclusivas (ET), em que o efeito não é incluído. Esta distinção entre operações IT e ET foi introduzida em [10].

Têm sido também identificadas várias propriedades de transformações [11] que ajudam a garantir a correcção dos algoritmos de OT. Destas destacam-se as propriedades de convergência, que serão discutidas a seguir. De qualquer forma, para decidir se estas propriedades devem ser garantidas ao nível do algoritmo de controlo de transformações ou pelas próprias transformações é necessário fazer um *tradeoff* entre a complexidade e dimensão a atribuir a cada camada arquitectural.

2.1.3. Propriedades de Transformações: Convergência

Em [11] são identificadas duas propriedades das funções de transformação, relativas ao problema de consistência. Estas propriedades são:

- **TP1** - $Oa \circ T(Ob,Oa) \equiv Ob \circ T(Oa,Ob)$ - A operação Oa composta com a transformação de Ob contra Oa é equivalente à operação Ob composta com a transformação de Oa contra Ob.
- **TP2** – $\forall O: T(T(O, Oa), T(Ob,Oa)) = T(T(O, Ob), T(Oa,Ob))$ – Para qualquer operação O, a sua transformação contra duas operações Oa e Ob deve dar sempre o mesmo resultado, siga que caminho seguir.

Estando discutidos os principais aspectos sobre OT em geral, pode-se agora ver as implementações referidas anteriormente.

2.1.4. Algoritmo GOTO

O algoritmo para controlo de operações GOTO [14] (GOT Optimized) é uma optimização do algoritmo de controlo GOT [12]. O GOT foi desenhado para ser usado em conjunto com o sistema REDUCE [10], de forma a garantir preservação de intenção.

O REDUCE só por si garantia preservação de causalidade, através de um esquema de vectores versão com estampilhas temporais, e convergência, através de um esquema de *undo/do/redo*. Permitia também que operações fossem executadas em qualquer ordem, desde que a sua causalidade fosse preservada. O que o esquema de *undo/do/redo* fazia, dada uma operação O causalmente pronta, era desfazer (*undo*) todas as operações já executadas que seguissem O, de forma a restaurar o documento ao estado anterior à execução destas, aplicar O (*do*) e refazer (*redo*) as operações anteriormente desfeitas. Desta forma era garantida convergência.

De forma a preservar intenção, o GOT garantia certas pré-condições das operações IT e ET e estas por sua vez garantiam certas pós-condições. Estas condições estavam relacionadas com os conceitos de contexto de definição (DC) e contexto de execução (EC) de uma operação. Contexto de definição é o conjunto de operações executadas até à definição de uma operação. Contexto de execução é o conjunto de operações executadas até à execução da operação.

- **IT(O1,O2):O1'** tinha como pré-condição que o DC de O1 fosse igual ao de O2 e como pós-condição o DC de O2 deveria preceder contextualmente o de O1'.

- **ET(O1,O2):O1'** tinha como pré-condição que o DC de O2 deveria preceder contextualmente o de O1 e como pós-condição o DC de O2 deveria ser igual ao de O1'.

Garantidas as suas pré-condições, se IT e ET garantissem as suas pós condições, o GOT era capaz de, dados uma operação O e o seu contexto de execução EC(O), transformar O em EO (forma de execução de O) através de IT e ET, de tal forma que DC(EO) = EC(O) (contexto de definição da forma de execução de O é igual ao contexto de execução de O). Se assim for, a execução de EO em EC(O) preserva a intenção de O.

A motivação para a criação do GOTO era pegar nas duas propriedades de convergência introduzidas pelo algoritmo adOPTed [11], TP1 e TP2, e introduzi-las nas operações de IT e ET usadas pelo algoritmo de controlo GOT. O que é demonstrado em [14] é que juntando TP1 e TP2 à lista de pós-condições de IT e ET do GOT e generalizando as noções de igualdade para equivalência nos conceitos de igualdade e precedência de contexto usados para definir DC e EC, o GOT original é capaz de assegurar consistência e preservação de intenção sem precisar do esquema *undo/do/redo* e sem usar um espaço multi-dimensional como o que é usado no adOPTed (em vez de usar um único vector para todas as operações como o REDUCE, o adOPTed usa um espaço *n*-dimensional, sendo *n* o número de participantes no documento). Para além disso, [14] demonstra que TP1 e TP2 podem ser usados para reduzir o número de operações IT e ET que é necessário executar. A este novo algoritmo optimizado foi dado o nome de GOTO.

2.1.5. Sistema Jupiter

O Jupiter [13] é um sistema desenhado para colaboração remota com diversas funcionalidades multimédia, incluindo edição colaborativa de documentos e comunicações áudio/vídeo. Como o sistema já possuía um servidor central para suportar estas funcionalidades, este foi aproveitado para implementar o seu algoritmo de controlo de concorrência optimística, baseado em Operational Transformation. O algoritmo criado parte do algoritmo do sistema GROVE [9], adaptando-o para comunicações entre clientes e servidor.

No Jupiter, todos os participantes num documento colaborativo mantêm uma réplica local, tal e qual como no GROVE. Diferentemente do GROVE, o servidor também mantém uma réplica e comunicações são sempre feitas entre um cliente e um servidor, não havendo comunicações entre os vários clientes. O que isto significa é que todas as comunicações são sempre *2-way*. Quando uma nova operação é gerada num cliente, este executa-a imediatamente na sua réplica e envia-a para o servidor. O servidor transforma-a contra a sua réplica, se necessário, executa-a e então faz *broadcast* da mesma para todos os outros clientes. Um cliente, ao receber uma operação do servidor, transforma-a se necessário e por fim executa-a. Estas comunicações *2-way* entre clientes e servidor, que formam uma topologia de comunicações em estrela, impedem que hajam problemas de causalidade, eliminando a preocupação em preservar causalidade no algoritmo de OT e simplificando-o substancialmente.

No entanto, o JUPITER necessita ainda de preservar convergência. Para tal, é usado um espaço de estados bidimensional para cada cliente, de forma a manter conta de todas as possíveis combinações de estados entre um cliente e o servidor. Para além disso, o algoritmo do JUPITER assegura que, para qualquer par de operações que estejam a ser transformadas, ambas devem ter sido geradas a partir do mesmo estado no espaço de estados bidireccional (que é o mesmo que a equivalência de contexto no GOT [12]). Juntando a estes dois mecanismos o facto de todas as comunicações serem apenas *2-way*, o JUPITER consegue não só manter consistência, como consegue resolver o problema que o GROVE [9] tem em certos casos específicos.

2.1.6. Google Wave Operational Transformation

A solução apresentada pelo Google Wave [15] parte da solução usada no JUPITER [13] e estende-a de forma a aumentar a sua eficiência. Tal como no JUPITER, é usada uma versão cliente-servidor do algoritmo de OT e um espaço de estados bidimensional para cada par cliente-servidor. O modelo de consistência usado, como no JUPITER, é o CC.

Em [15] é identificado um problema de escalabilidade no JUPITER. Como o JUPITER mantém no servidor um grafo bidimensional para cada cliente ligado a este, o número de possíveis clientes que podem usar o sistema ao mesmo tempo é afectado drasticamente. Por outro lado, o facto de haver um grafo para cada cliente também complica o algoritmo do servidor, ao requerer que este converta operações de clientes entre grafos.

Para resolver este problema o Google Wave introduz um mecanismo de *stop'n'wait* e *acknowledgments* no envio de novas operações de um cliente para o servidor. Ao gerar uma nova operação, o cliente envia-a para o servidor e fica à espera do seu *acknowledgment*, em vez de continuar a enviar novas operações como acontecia no JUPITER. O servidor, ao receber a operação, transforma-a se necessário, executa-a na sua réplica e envia-a para todos os outros clientes ligados. Depois disto o servidor envia o *acknowledgment* para o cliente que gerou a operação e este pode então enviar mais operações. Enquanto espera, o cliente vai fazendo *caching* de novas operações e depois envia-as em massa.

Com este novo mecanismo, o cliente passa a poder inferir o caminho do servidor no grafo bidimensional de estados e pode enviar sempre operações que fiquem no mesmo. Uma consequência importante é que o servidor passa a apenas precisar de um grafo de estados, que não é mais que o historial de operações executadas. Operações de clientes são guardadas segundo a sua data de recepção, de forma a manter causalidade. As operações podem ser recebidas concorrentemente e, se assim for, o servidor decide a ordenação a impor e usa o algoritmo descrito para manter causalidade.

É de notar que como todos os clientes recebem as operações já aplicadas pelo servidor na mesma ordem, irão concordar na ordem das operações remotas. Esta ordenação global conseguida através do servidor permite que o algoritmo de OT do Google Wave não tenha que implementar TP2. No entanto, TP1 é implementada, como na generalidade dos algoritmos de OT. Segue-se um exemplo que pretende demonstrar que o algoritmo de OT do Google Wave não precisa de implementar TP2. Dado um servidor com espaço de estados igual a [O] e três clientes que pretendem enviar uma operação cada um, ou seja:

1. Espaço de Estados do Cliente A: [O, Oa]
2. Espaço de Estados do Cliente B: [O, Ob]
3. Espaço de Estados do Cliente C: [O, Oc]

O servidor decide aplicar as operações recebidas concorrentemente por ordem alfabética, ou seja, o seu espaço de estados passa a ser: [O, Oa, Ob', Oc'], com Ob' = T(Ob, Oa) e Oc' = T(T(Oc, Oa), T(Ob, Oa)). Ao receber as operações aplicadas pelo servidor, os clientes transformam-nas de forma a incluir o efeito de operações locais que ainda não foram *acknowledged*, isto é, usam a operação IT. Assim sendo os clientes ficam com os seguintes espaços de estados:

1. Cliente A: [O, Oa, Ob', Oc'], com Ob' = T(Ob,Oa) e Oc' = T(T(Oc, Oa), T(Ob, Oa))
2. Cliente B: [O, Ob, Oa', Oc'], com Oa' = T(Oa,Ob) e Oc' = T(T(Oc, Oa), T(Ob, Oa))
3. Cliente C: [O, Oc, Oa", Ob"], com Oa" = T(Oa,Oc) e Ob" = T(Ob', T(Oc, Oa))

O cliente A tem o mesmo espaço de estados que o servidor, ao contrário dos clientes B e C. No entanto, eles chegarão ao mesmo estado final de documento que o servidor, apesar de

seguirem caminhos diferentes. Usando TP1 (ver definição no capitulo 2.1.3), é possível demonstrar essas conclusões:

Cliente B: [O, Ob, Oa', Oc'] ~ [O, Ob, T(Oa, Ob), Oc']
~ [O, Oa, T(Ob, Oa), Oc'] (uso de TP1)
~ [O, Oa, Ob', Oc']

Cliente C: [O, Oc, Oa'', Ob''] ~ [O, Oc, T(Oa, Oc), Ob'']
~ [O, Oa, T(Oc, Oa), T(Ob', T(Oc, Oa))] (uso de TP1)
~ [O, Oa, Ob', T(T(Oc, Oa), Ob')] (uso de TP1)
~ [O, Oa, Ob', T(T(Oc, Oa), T(Ob, Oa))]
~ [O, Oa, Ob', Oc']

De forma a aumentar ainda mais a escalabilidade, o Google Wave usa mais dois mecanismos: um para recuperação de falhas de comunicação e do servidor e outro para recuperação de erros nos documentos XML. Este último mecanismo é conseguido através da comunicação de *checksums* dos documentos XML sempre que é transmitida uma operação ou *acknowledgment*.

Por fim, de modo a resolver conflitos de forma eficiente quando cliente e servidor estão muito dessincronizados, o Google Wave usa um mecanismo de transformação e composição de operações, baseado em streaming de operações e na definição de uma operação como sendo um conjunto de mutações de um documento XML. Através deste mecanismo, demonstrado em [15], o Google Wave é capaz de resolver N x M conflitos (N é o número de operações que o servidor ainda não fez *acknowledge* e M o número de operações que o cliente ainda não fez *acknowledge*) em $O(N \times \log(N) + M \times \log(M))$, em vez de $O(N \times M)$, como seria de esperar de algoritmos de OT tradicionais.

2.1.7. Sumário

Nesta secção foi apresentado o algoritmo optimístico para controlo de concorrência conhecido como Operational Transformation. O algoritmo é relevante para este trabalho pois é a tecnologia chave do Wave, permitindo edição colaborativa de documentos em tempo quase real. Concretamente, foram analisados os vários pormenores da tecnologia como modelos de consistência, arquitectura e propriedades de transformações, de forma a fornecer uma base sobre o tema. Depois foram discutidas diferentes aproximações, incluindo o sistema REDUCE [10] e algoritmos GOT [12], GOTO [14] e adOPTed [11], sistema JUPITER [13] e, por fim, a aproximação do Wave [15]. Esta baseou-se fundamentalmente no trabalho do sistema JUPITER, no entanto apresentou-se como uma versão melhorada, em termos de escalabilidade e performance do servidor.

Tendo em conta os objectivos da dissertação, bem como as contribuições desenvolvidas, da análise anterior resulta que dos sistemas estudados se tornou particularmente importante ter em conta os sistemas GOTO e JUPITER nas seguintes dimensões: o sistema GOTO, por ser dos sistemas estudados o mais completo, em termos de modelo de consistência usado e equilíbrio conseguido entre complexidade das diferentes camadas arquitecturais, e por ser um sistema completamente distribuído; o sistema JUPITER por ser o único sistema centralizado conhecido e por servir de base ao algoritmo do Wave.

O estudo realizado revelou-se especialmente importante para a contribuição da interface portável para o Wave, de forma a compreender como eram processadas as operações pelo servidor Wave. O estudo revelou-se também importante para a contribuição de adição de edição colaborativa de documentos e integração do Wave na plataforma, por ser a tecnologia chave por traz dos mesmos.

2.2. Plataformas e Protocolos de Colaboração

Nesta secção vão ser analisados protocolos de colaboração, de serviços de presença e de *instant messaging*. Nomeadamente, vão ser discutidos os protocolos SIP, SIMPLE e XMPP e vai ser apresentada e discutida a solução usada pelo Google Wave.
Vão também ser discutidas plataformas colaborativas, nomeadamente a plataforma colaborativa que está a ser desenvolvida na PT Inovação, o PUC.

2.2.1. SIP

SIP [23] (*Session Initiation Protocol*) é um protocolo de nível aplicacional para sinalização em redes IP que permite criar, modificar e manter sessões multimédia com um ou mais participantes. O SIP trabalha em conjunto com vários protocolos de multimédia, permitindo que *endpoints* na internet (user agents) descubram-se uns aos outros e concordem nos detalhes da sessão que pretendem estabelecer. De modo a auxiliar a descoberta de user agents, o SIP permite também a criação de "pontos de encontro", ou seja, infra-estruturas na rede (proxy servers) onde os user agents se podem registar, enviar convites para sessões, etc. Por outro lado o SIP é independente dos protocolos de transporte usados por baixo, sendo compatível com UDP, TCP e SCTP, o que faz dele um protocolo ágil e de carácter geral.

O SIP não oferece serviços, mas sim primitivas que podem ser usadas para implementar vários tipos de serviços. Assim sendo, o SIP não pode ser visto como sistema integrado para comunicações mas sim como um componente que pode ser usado em conjunto com outros protocolos de multimédia de forma a construir uma arquitectura multimédia completa. O SIP suporta cinco primitivas principais:

- **Localização de Utilizadores:** determinação do *endpoint* a ser usado para a sessão;
- **Disponibilidade de Utilizadores:** determinação da vontade de participar na sessão por parte da entidade que está a ser convidada;
- **Capacidade de Utilizadores:** determinação dos media a usar e parâmetros sobre os mesmos;
- **Configuração de Sessões:** estabelecimento de parâmetros em todas as entidades envolvidas na sessão;
- **Gestão de Sessões:** transferência e terminação de sessões, modificação de parâmetros, invocação de serviços, etc.

Mantido actualmente pelo Internet Engineering Task Force (IETF), o SIP é um *open standard* com vários RFCs propostos, sendo [23] o principal. Pretendendo tirar partido da sua agilidade e carácter geral, têm surgido várias extensões do SIP, das quais se destaca uma para serviços de IM e presença, o SIMPLE.

2.2.2. SIMPLE

O SIMPLE [21] (*Session Initiation Protocol for Instant Messaging and Presence Leveraging Extensions*) é um protocolo para IM e presença baseado no SIP. Como no caso do SIP, existe também no IETF um *workgroup* a trabalhar no SIMPLE. No entanto na altura da escrita deste relatório ainda tinham sido produzidos poucos RFCs e *drafts* pelo mesmo [21]. Como o SIP, o SIMPLE é também um *open standard*.

O SIMPLE apresenta como grandes vantagens, face a outros protocolos de IM, suporte para entrega assíncrona de mensagens, uso de TLS (*Transport Layer Security*) e número ilimitado de contactos. Por outro lado, como as funcionalidades de presença e IM que oferece não são muito complexas e como é baseado no SIP, que foi desenhado de raiz de modo a ser simples e genérico, o SIMPLE permite processar eventos de forma rápida. Isto é particularmente importante para redes como IMS (IP *Multimedia Subsystem*), onde o uso de SIP também impera.

No entanto, o SIMPLE também apresenta graves desvantagens. Em primeiro lugar, como o processo de estandardização do protocolo ainda está pouco avançado, existem poucas implementações de serviços de presença baseados no mesmo e, consequentemente, poucas aplicações cliente. Em segundo lugar, segundo [22] o SIMPLE sofre de graves problemas de escalabilidade devido à grande dimensão e quantidade de pacotes trocados entre domínios, o que provoca sobrecarga da largura de banda disponível para as aplicações. Tal acontece devido a basear-se num protocolo que foi desenhado para criação de sessões, onde o número e dimensão das mensagens têm pouca importância, ao contrário de sistemas de presença e IM onde estes problemas têm elevada importância. A proposta final do SIMPLE *workgroup*, depois de considerar várias optimizações, é que o desenho do protocolo deve ser mudado no seu *core* de forma a endereçar o problema da escalabilidade ou, em vez de usar o mesmo protocolo para comunicações C/S e S/S, deve ser usado um protocolo diferente para comunicações entre servidores (S/S). Este novo protocolo deve ser desenhado de forma a optimizar a carga da rede, usando obrigatoriamente o protocolo de transporte TCP em vez de UDP, por exemplo.

2.2.3. XMPP

XMPP [16] é um protocolo extensível, baseado em XML, para *Instant Messaging* em tempo quase real e serviços de presença. Inicialmente criado como um único protocolo, o XMPP evoluiu desde então para um vasto conjunto de protocolos e *drafts* sobre não só IM e serviços de presença, como serviços de *publish*/*subscribe* e *Message Oriented Middleware*. Estes protocolos dividem-se em duas categorias: os centrais (core) [19], que foram publicados pela IETF, e as extensões oficiais, publicadas pela XMPP *Standards Foundation* (XSF).
Ao contrário da maior parte dos protocolos deste estilo e como o SIP/SIMPLE, o XMPP é um *open standard*. Isto significa que qualquer pessoa ou entidade com um domínio pode hospedar um servidor XMPP e comunicar com utilizadores de outros servidores.

Este princípio de *open standards*, muito como os E-Mails e o protocolo SMTP, é também procurado pelos criadores do Google Wave. Daí que o protocolo de federação do Wave, que é usado para comunicação entre os seus servidores, tenha sido construído como uma extensão ao XMPP.
Analisando em detalhe o XMPP, vemos que este possui os seguintes pontos fortes:

- **Descentralização**: qualquer pessoa ou entidade pode correr um servidor e não há um servidor central que controle todos os outros;
- **Segurança**: tecnologias de segurança, como SASL e TLS, foram implementadas de raiz no próprio core do XMPP, sendo também possível isolar completamente um servidor do público e filtrar o seu acesso;
- **Open Standards**: os protocolos XMPP foram publicados pelo IETF como RFCs e podem ser livremente implementados por qualquer terceiro, não sendo necessárias quaisquer regalias.
- **Flexibilidade**: o XMPP pode ser estendido livremente e novas funcionalidades podem ser criadas em cima do mesmo.

Comparando o XMPP com o SIMPLE, o primeiro tem a vantagem de ser um protocolo já devidamente estandardizado e com uma forte comunidade *open source* por traz. Por outro lado, apesar de não estar ligado a nenhuma arquitectura específica, ele foi desenhado e é geralmente usado juntamente com TCP, melhorando o desempenho da largura de banda usada e escalabilidade.

A figura 2 mostra um exemplo simples do protocolo em funcionamento [19]. O protocolo começa com a abertura de um XML *Stream* por parte do cliente. Como cada *Stream* é unidireccional, o servidor responde abrindo outro *Stream* para o cliente. Basicamente, um XML *Stream* é um contentor através do qual podem ser trocadas mensagens de autenticação, de presença, IM, etc. Estas mensagens de presença e IM são conhecidas como XML *Stanzas* e podem ser de três tipos: simples envio de informação (*message*), *publish*/*subscribe* de informação (*presence*) e *info*/*query* de informação (*iq*).

Exemplo disto é a mensagem enviado pelo cliente na figura 2, depois de terminado o processo de autenticação. A mensagem é recebida pelo servidor e enviada para o destino respectivo, que neste caso pertence ao mesmo domínio. Caso pertence-se a outro domínio, a mensagem era primeiro enviada para o seu servidor e então para o cliente destino. Comunicação entre servidores XMPP segue a mesma metodologia já descrita, com criação de XML *Streams*, etc. Para terminar a sessão, é enviada uma mensagem de fim de *Stream* pelo cliente, à qual o servidor responde fechando o outro *Stream* criado por ele.

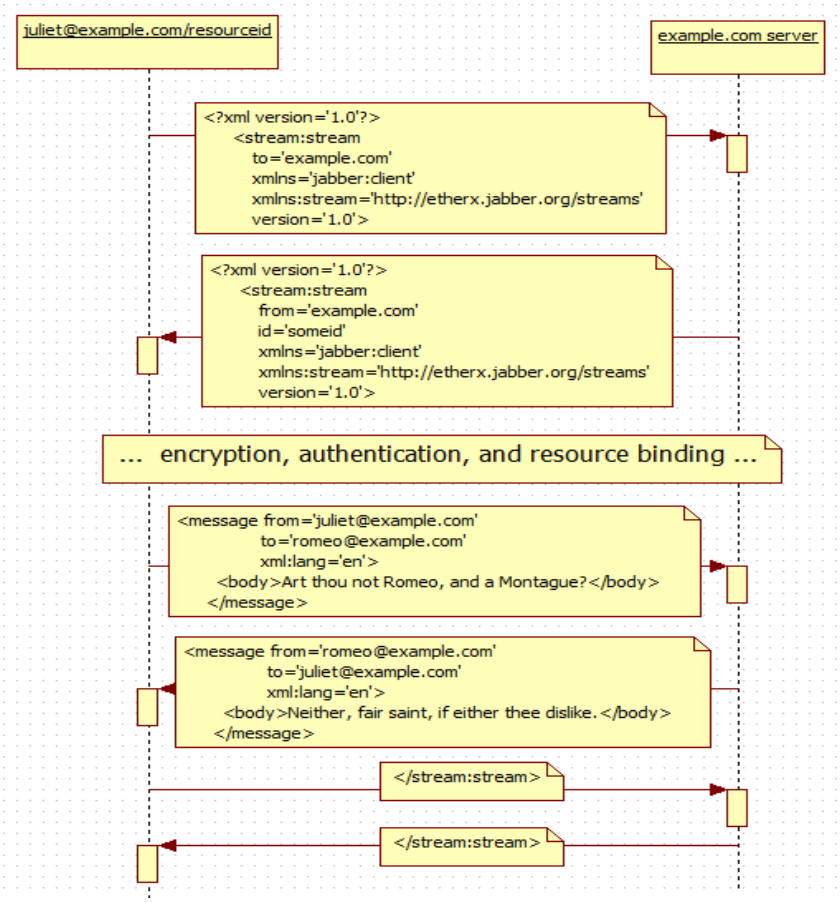


Figura 2 - Exemplo de sessão XMPP

2.2.4. Google Wave Federation Protocol

O protocolo de federação do Google Wave [20], criado para estabelecer as comunicações entre servidores Wave, é uma extensão aberta do protocolo XMPP Core [19]. Como tal, é um protocolo que permite comunicação de alterações de documentos entre servidores em tempo quase real. Enquanto extensão, o protocolo de federação adiciona novos tipos de mensagens, necessárias para as operações e funcionalidades do Google Wave, que são suportadas pelas mensagens XMPP já existentes.

Uma grande diferença entre o protocolo XMPP e o Google Wave é que enquanto no XMPP Core as mensagens têm como destino um utilizador num dado domínio, onde haverá um servidor XMPP a funcionar, no Google Wave e no seu protocolo os destinos das mensagens são as Waves, isto é, os utilizadores não escrevem directamente entre si mas escrevem num documento que é visto e partilhado por todos.

O protocolo de federação do Google Wave usa, para além do XMPP Core, uma extensão oficial do XMPP, descrita em [24]. Esta extensão explica como implementar modelos de *publish*/*subscribe* através do modelo de mensagens do XMPP e, como tal, permite aumentar

ainda mais o nível de assincronia do Google Wave, para além do conseguido com a extensão do XMPP Core. Consequentemente, este aumento de assincronia causa também um aumento de escalabilidade, pois desacopla os vários componentes da arquitectura.

2.2.5. PUC

O PUC [3] (Plataforma Unificada de Colaboração) é uma plataforma para colaboração que está a ser desenvolvida na PT Inovação. Esta plataforma é a base de todo o trabalho realizado nesta dissertação. Como tal, a plataforma desenvolvida partilha a mesma arquitectura e modelo de dados que esta, assim como funcionalidades básicas.

O PUC possui a arquitectura resumida na figura 3. A figura apresenta uma arquitectura 3-tier, com o nível mais baixo dos dados/recursos, o nível intermédio dos controladores de recursos (*Service Enablers*) e o nível mais alto das aplicações que usam as APIs expostas pelo nível intermédio. Nesta arquitectura, os componentes que compõem a plataforma em si correspondem à camada intermédia.

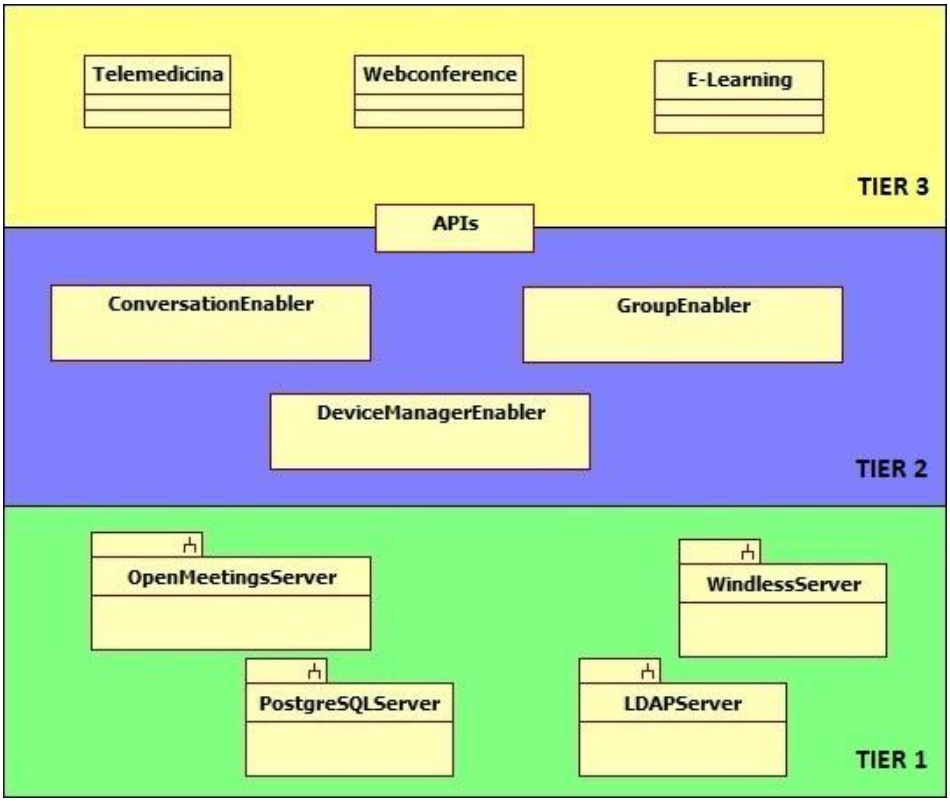


Figura 3 - Arquitectura 3-tier do PUC

Como foi dito, a camada superior da arquitectura é composta por aplicações colaborativas complexas, com objectivos específicos e que fazem uso dos serviços colaborativos

disponibilizados pelo PUC. Como exemplos temos aplicações de Telemedicina, Webconference, E-Learning, etc.

Ainda segundo a figura 3, o PUC possui dois grandes recursos que lhe permitem oferecer uma série de funcionalidades colaborativas. Estes são o OpenMeetings [25] e o IP-Windless. O OpenMeetings é uma plataforma para *WebConferencing* construída sobre um servidor *open-source* de flash, o Red5 [26]. A utilização do OpenMeetings permite integrar no PUC serviços baseados em *streaming* de áudio e vídeo: *WhiteBoarding*, *WebConferencing*, *Slideshow* e partilha de ficheiros. O IP-Windless é um Media Server IP genérico, desenvolvido na PT Inovação, que disponibiliza uma vasta gama de funcionalidades de processamento de diferentes tipos de media: tocar/gravar anúncios de áudio e vídeo, reconhecimento de fala, síntese de texto para fala, conferências de áudio e vídeo, fax, etc.

Como plataforma unificada de colaboração, o objectivo do PUC é unir estes diferentes recursos colaborativos sobre uma só interface. Esta interface é usada pelas aplicações para invocarem os serviços colaborativos a oferecer aos utilizadores. Para comunicarem com a plataforma as aplicações usam diferentes topologias de redes e são executadas a partir de dispositivos heterogéneos, incluindo PCs, *smartphones*, televisores, etc. Uma das vantagens desta aproximação é o nível de abstracção conseguido entre as aplicações e a camada de recursos. Para conseguir tal abstracção a plataforma possui vários *Service Enablers*, módulos independentes que implementam um conjunto de funcionalidades e que permitem controlar os vários recursos disponíveis no sistema. Cada *Service Enabler* tem uma função distinta e expõe um conjunto de interfaces separado. Os *Service Enablers* são:

- **Device Enabler:**

Este módulo possibilita gerir detalhes de dispositivos específicos e suas capacidades, de forma a abstrair e aumentar a portabilidade do PUC. O módulo está a ser desenvolvido em paralelo com esta dissetação [41].

- **Group Enabler:**

Este módulo é responsável pela gestão e contextualização dos dados relativos à informação pessoal dos utilizadores da plataforma. Os vários contactos do utilizador (número de telefone, email, *instant messaging*, SIP URI, etc) são tratados de uma forma heterogénea, permitindo que se adicionem novos contactos à medida que se adicionam mais funcionalidades à plataforma. Cada um destes endereços corresponde a um tipo de recurso e tem associado um estado de presença (Disponível, Ausente, Ocupado, etc) utilizado para construir o contexto agregado do utilizador. O Group Enabler tem também como o próprio nome indica a responsabilidade de efectuar a gestão dos grupos criados através da plataforma, sejam listas de contactos de utilizador (*buddy lists*) ou listas de participantes de uma determinada sessão ou conversa.

- **Conversation Enabler:**

Este é o módulo que permite gerir as sessões colaborativas da plataforma. Ao contrário dos anteriores, este *Service Enabler* expõe não uma mas várias interfaces para as aplicações clientes. Estas estão ligadas aos componentes mais importantes do modelo de dados da plataforma: Conversa e Sessão. A conversa é a unidade principal de abstracção na plataforma. É constituída por um grupo, que inclui os utilizadores a si associados, e por um conjunto de sessões. Uma sessão define-se como uma parte de uma conversa ocorrida num determinado intervalo de tempo onde se discute um certo tópico. A plataforma permite que haja vários tipos de sessão (videoconferência, chat, *slideshow*, etc), dependendo dos servidores de recursos disponíveis e das funcionalidades que se pretenda utilizar. A cada sessão está associado um grupo (que identifica os participantes da sessão e é um subconjunto do grupo da conversa) e vários *ResourceFeatures*, que representam os vários recursos partilhados na sessão.

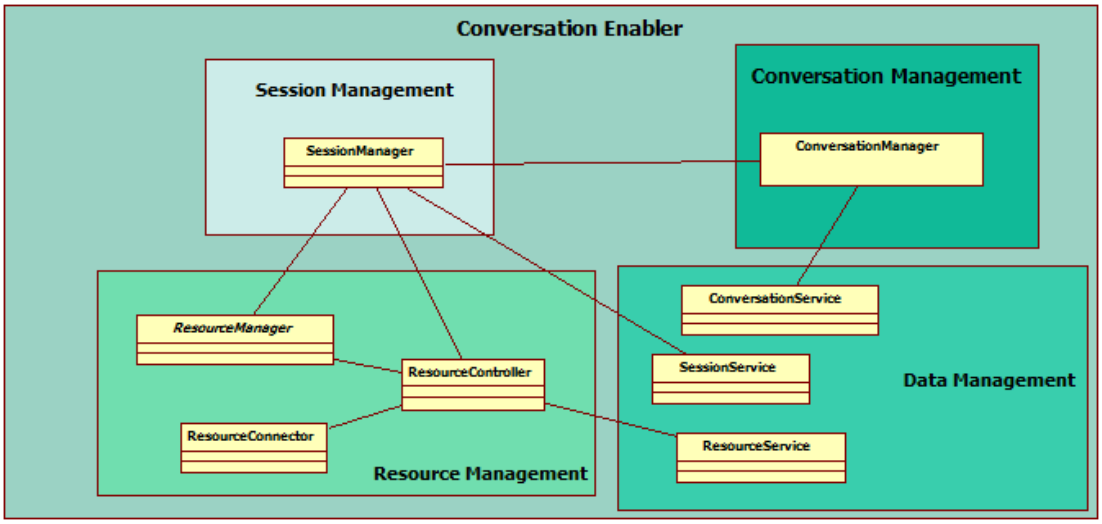


Figura 4 - *Conversation Enabler* do PUC

A figura 4 mostra o *Conversation Enabler* e seus componentes. De forma a possibilitar que as aplicações possam gerir as diferentes entidades de uma conversa, o *Conversation Enabler* disponibiliza as duas interfaces referidas:

- **Conversation Manager:** Possibilita gerir todo o conjunto de conversas existentes no sistema, assim como criar uma nova conversa ou apagar uma já existente e controlar os detalhes específicos de cada uma.
- **Session Manager:** Faculta gestão e controlo de todas as sessões da plataforma e seus componentes, incluindo a adição ou remoção de participantes, a actualização do estado de uma sessão, etc.

De forma a oferecer as suas funcionalidades às aplicações, estes componentes usam outros de mais baixo nível que também fazem parte do *Conversation Enabler*: os componentes de *Resource Management*, que permitem gerir os vários tipos de recursos, e os de *Data*

Management, que permitem gerir e persistir os vários tipos de dados (dados de recursos, de sessões e de conversas).

O grande objectivo do PUC é fornecer uma interface colaborativa genérica para agendamento e gestão de sessões colaborativas. No entanto, e de forma aumentar a performance das aplicações que o usam, o PUC permite que estas interajam directamente com os recursos, de forma a efectuar operações computacionalmente mais pesadas. Para tal o PUC oferece interfaces portáveis para que aplicações em dispositivos heterogéneos possam fazer a comunicação directa com os recursos.

2.2.6. Sumário

Nesta secção foram vistos diversos protocolos de colaboração e de presença. Mais concretamente foi visto o protocolo de federação do Google Wave, usado para comunicações entre servidores, e o protocolo XMPP, no qual o primeiro se baseia e estende. Foram ainda vistos os protocolos SIP e SIMPLE, como base comparativa com o XMPP e protocolo de federação. A conclusão inferida do estudo é que o uso de XMPP como base para o protocolo de federação traz grandes vantagens, principalmente em termos de escalabilidade e extensibilidade.

Foi também analisada em pormenor a plataforma colaborativa que serve de base para todo o trabalho realizado, o PUC. Do que foi discutido conclui-se que é uma plataforma colaborativa genérica e bem modularizada. Tal foi importante pois permitiu desenvolver as contribuições desta dissertação sem ter que alterar toda a sua estrutura interna.

Do estudo realizado foi particularmente importante a análise feita aos protocolos colaborativos para as contribuições da interface portável para o Wave e para a integração do Wave na plataforma. O estudo da plataforma PUC foi de extrema importância para todas as contribuições desenvolvidas.

2.3. Serviços Colaborativos

Nesta secção vão ser apresentados diferentes exemplos de serviços colaborativos, assim como vão ser analisadas as tecnologias e protocolos por eles usados. Para cada serviço estudado será também dito como se relaciona com este trabalho.

2.3.1. Google Docs

Google Docs [17] é uma ferramenta da Google para edição colaborativa de documentos em tempo semi-real. Baseado na ideia de software como serviço, o Google Docs pode ser usado através de qualquer browser que permita Cookies e JavaScript. Pode também ser usado em modo *offline*, através do Google Web Toolkit (GWT).

O Google Docs é composto por três módulos principais: *Documents*, *Spreadsheets* e *Presentations*. Isto significa que pode ser usado para editar colaborativamente documentos de *rich-text*, folhas de cálculo e apresentações. Para permitir colaboração em tempo semi-real, o Google Docs mantém os documentos criados nos seus servidores, permitindo também assim persistência. Actualizações são feitas geralmente de 30 em 30 segundos [18]. O protocolo usado é baseado em HTTP e XML. O Google Docs possui também uma API que permite estender a aplicação cliente.

O Google Docs relaciona-se com o sistema implementado por ser um bom exemplo de um serviço para edição colaborativa de documentos de texto rico, para além de guardar persistentemente os documentos e poder ser acedido por um browser.

2.3.2. CoOffice

CoOffice [27] é um conjunto de ferramentas criadas de forma a tornar aplicações *single-user* em aplicações colaborativas, nomeadamente o Word e PowerPoint da família de produtos Office da Microsoft. Assim sendo, o CoOffice é um serviço que permite edição colaborativa de documentos de texto rico, através do CoWord, e de apresentações através do CoPowepoint.

Para transformar aplicações *single-user* em aplicações colaborativas, os autores do CoOffice desenvolveram um mecanismo de adaptação transparente (*Transparent Adaptation* [27]). O que este mecanismo faz, como o próprio nome indica, é adaptar a aplicação que se pretende tornar colaborativa, sem qualquer alteração da mesma. Para tal, esta deve disponibilizar uma API que permita adaptar o seu modelo de dados para colaboração. De modo a efectuar a adaptação, o mecanismo usa a API para interceptar as interacções locais do utilizador, converte-as em operações abstractas, manipula-as usando técnicas colaborativas e por fim interpreta as operações modificadas nos outros sites colaborativos, através da API.

Esta aproximação tem a vantagem de, ao obrigar que as aplicações sejam adaptadas, permitir o uso de conhecimentos semânticos das mesmas e uso de controlo de concorrência optimístico. Tirando partido disto, o método de controlo de concorrência usado no CoOffice é Operational Transformation (analisado em detalhe no capítulo 2.1).

O algoritmo de OT usado baseia-se no sistema REDUCE [10, 28] (estudado no capítulo 2.1.5), que é dos mesmos autores, suportando consistência e *group undo* [28]. De modo a poder-se utilizar OT num editor complexo de *rich-text* como o Word, quando este normalmente é usado em simples editores de texto, o CoOffice estende o modelo de dados básico de OT (um espaço de endereçamento singular e linear) para uma árvore de grupos de endereçamento. Dentro de cada um destes grupos existem múltiplos espaços de endereçamento lineares. As folhas da árvore são objectos terminais (como letras e números). Usando este modelo de dados, um simples documento de texto, por exemplo, ficaria representado como uma árvore singular com um único grupo, possuindo este por sua vez um único espaço de endereçamento. O CoOffice adiciona também uma nova operação básica de *update* às operações usadas normalmente, *insert* e *delete*. Com estas três operações de OT consegue-se mapear todas as operações de alto nível do Word e PowerPoint, segundo [27].

O CoOffice relaciona-se com o sistema implementado porque é um bom exemplo de editor de texto rico colaborativo, baseado em OT, e que usa uma aproximação diferente dos habituais sistemas de OT.

2.3.3. Cisco WebEx

Cisco WebEx [29] é uma empresa que oferece uma série de produtos e serviços de colaboração e conferências Web. Geralmente esta família de soluções é apenas referida como WebEx. Entre as principais funcionalidades oferecidas pelos diferentes produtos WebEx, listam-se:

- Partilha de documentos, apresentações, ficheiros e outros tipos de média;
- *Webconferencing* em tempo real e com múltiplos utilizadores;
- *Whiteboarding* e outras funcionalidades de *Brainstorming*;
- Chat privado entre utilizadores;
- Partilha de desktop (*DesktopSharing*);
- Partilha de aplicações (*ApplicationSharing*);
- Partilha de browser e de conteúdo Web;
- Controlo de utilizadores e delegação de controlo e responsabilidades;

Os produtos WebEx são desenvolvidos sobre a Web e podem ser usados em qualquer browser, dependendo apenas da existência de uma Java Virtual Machine. Partilha de documentos

e outros média é feita de forma distribuída e sem uso de servidores intermédios, o que reduz riscos e preocupações de segurança no armazenamento de documentos confidenciais.

Para implementar as suas diferentes funcionalidades, o WebEx usa o conjunto de protocolos T.120 [30], que definem uma série de serviços para suporte de comunicações multiponto em tempo quase real. Segundo [31], as vantagens mais importantes que o standard T.120 oferece são:

- Entrega de dados multiponto fiável, com uso de Multicast se disponível na rede;
- Interoperabilidade, Transparência e Independência de Rede e Independência Aplicacional e de Plataforma;
- Escalabilidade e Extensibilidade.

Ao implementar o standard T.120, o WebEx consegue tirar partido de todos os benefícios listados, entre outros.

O WebEx oferece três grandes APIs que permitem integrar os seus serviços noutras aplicações. Estas APIs são: URL API, XML API e *Teleconference Service Provider* (TSP) API. A URL API, como forma de integração mais simples, é ideal para integração leve em portais Web ou outras aplicações baseadas na Web. Esta API permite agendar, começar ou juntar-se a reuniões, por exemplo. A URL API é baseada em pedidos HTTP que invocam serviços no WebEx. Tais pedidos são feitos aos servidores WebEx, através de urls especificamente construídos para cada funcionalidade. Os pedidos são processados e é retornada uma página Web específica como resultado da operação.

Para integrar actividades mais avançadas do WebEx a nível aplicacional existe a XML API. Esta API envolve a troca de mensagens XML entre aplicações e servidores WebEx, para pedidos das aplicações aos servidores e respostas no sentido oposto. Funcionalidades conseguidas através da XML API vão desde a criação de utilizadores até à troca de ficheiros.

A TSP API, dirigida a vendedores de redes de conferências, permite que estes integrem os seus serviços telefónicos com os serviços WebEx, de modo a formar um ambiente ininterrupto de conferência áudio/vídeo.

O Cisco WebEx relaciona-se com o sistema implementado porque é um bom exemplo de um serviço de colaboração avançado, que oferece várias funcionalidades colaborativas como conferências Web, *Whiteboarding*, entre outras, e que no entanto usa uma aproximação diferente para implementar as suas funcionalidades.

2.3.4. Novell Pulse

O Novell PULSE [32] é um novo produto da Novell, cujo lançamento é esperado para meio de 2010. O PULSE é um serviço colaborativo em tempo quase real, desenhado de raiz para ser compatível com o Google Wave, ou seja, o Pulse implementa o protocolo de federação do

Google Wave. No entanto, o Pulse tem como alvo o mundo empresarial, fazendo parte de um novo conceito tecnológico que é a Enterprise 2.0.

Como o Wave, o Pulse combina e-mail, edição colaborativa de documentos e serviços de mensagens e presença. Porém, a estas funcionalidades do Wave o Pulse adiciona funcionalidades chave para as empresas, como segurança e gestão de identidades.

O Novell Pulse relaciona-se, por um lado, com o sistema concebido porque é um exemplo de um serviço colaborativo em tempo quase real dirigido ao mundo empresarial. Por outro, o Novell Pulse oferece as mesmas funcionalidades do Google Wave e implementa o seu protocolo de federação, ou seja, é compatível com o Google Wave.

2.3.5. Google Wave

O Google Wave [5] é um serviço para colaboração lançado pela Google. Na altura de escrita deste relatório, o Google Wave encontra-se em fase de preview.

No Google Wave uma sessão colaborativa chama-se Wave. Uma Wave pode ter várias sub-sessões, denominadas Wavelets. Uma Wavelet é composta por um conjunto de participantes e por um conjunto de documentos, denominados Blips. A Wavelet é o elemento principal do modelo de dados do Google Wave e é ao seu nível que é aplicada a técnica de Operational Transformation (analisada em detalhe no capítulo 2.1).

O Google Wave possui várias funcionalidades, das quais a principal é edição colaborativa de documentos de texto rico em tempo quase real. Tal é conseguido usando um algoritmo centralizado de Operational Transformation, como foi discutido no capítulo 2.1.7. De entre as outras funcionalidades oferecidas pelo Google Wave, destacam-se:

- Interface Web, que pode ser acedida em qualquer browser com suporte para HTML 5 e GWT (Google Web Toolkit);
- Partilha de ficheiros através de *drag and drop*, com especial ênfase na partilha de imagens;
- Visualização de imagens partilhadas em *slide show*;
- Reprodução do histórico de alterações efectuadas numa Wavelet;
- Imbricação de documentos dentro de documentos e de Wavelets dentro de Wavelets;
- Persistência dos documentos colaborativos e dos ficheiros partilhados nos mesmos;
- Grande nível de extensibilidade, com dois conjuntos de APIs [8] que permitem implementar um número quase ilimitado de novas funcionalidades.

Das APIs para extensibilidade referidas destacam-se a Robot e Gadget APIs. A primeira permite criar pequenas aplicações que são inseridas nas Wavelets como participantes e podem interagir com os utilizadores do serviço. A segunda permite integrar no Wave *widgets*

colaborativas, possibilitando um leque quase ilimitado de funcionalidades dirigidas para a colaboração.

De modo a ter a maior aceitação pública possível e de modo a tornar-se uma norma vastamente utilizada o Wave foi lançado como uma norma aberta. Neste sentido foram disponibilizados os vários protocolos e especificações usados para implementar o serviço Google Wave, incluindo [15], [20] e [60], entre outros. Foi também disponibilizado, como projecto *open source*, um servidor Wave de protótipo denominado de FedOne [4]. No entanto este servidor apenas possui funcionalidades de edição colaborativa de documentos e não implementa completamente todas as especificações Wave.

Como no E-Mail, em que qualquer pessoa ou entidade pode possuir um servidor SMTP, também qualquer pessoa ou entidade pode possuir um servidor Wave e ter o seu próprio domínio. Isto significa que o Google Wave não tem um ponto central de falha, pois haverá servidores Wave espalhados por todo o mundo a comunicarem entre si. Por outro lado, é necessário um protocolo para sincronização de versões entre servidores. Este protocolo é o Google Wave Federation Protocol e foi discutido no capítulo 2.2.4.

O Google Wave relaciona-se com esta dissertação porque foram exactamente os seus conceitos e tecnologias que serviram de inspiração para este trabalho e foram incorporados na plataforma desenvolvida.

2.3.6. Sumário

Nesta secção foram vistos vários exemplos de serviços colaborativos relacionados, de alguma forma, com este trabalho. A próxima tabela apresenta uma comparação de funcionalidades e requisitos arquitecturais dos serviços analisados, incluindo referências para outros cuja análise detalhada foi omitida de forma a não sobrecarregar este capítulo.

Tabela 1: Comparação de requisitos funcionais e arquitecturais

Descrição	Google Wave	Google Docs	CoOffice	Cisco Webex	Microsoft Office Live Communicator	Adobe Connect Pro	Novell Pulse
Controlo de Acesso às Conversas (públicas, privadas, etc) Pesquisa, Follow/Unfollow de Conversas	X						X
Classificação das Conversas / Sessões (eg tags)	X						X
Suporte para o estabelecimento de sessões de video-conferência	(possível através de gadgets)			X	X	X	
Suporte para o estabelecimento de sessões de comunicação áudio (VoIP)	(possível através de gadgets)			X	X	X	
Edição colaborativa de documentos	X	X	X	X	X	X	X

Descrição	Google Wave	Google Docs	CoOffice	Cisco Webex	Microsoft Office Live Communicator	Adobe Connect Pro	Novell Pulse
Import/Export de Docs para Edição incluindo PDF, Doc Office, OpenOffice, ..	(possível através de robots)	X	X		X		X
Apresentações	(possível através de gadgets)	X	X	X	X	X	
Partilha de fotos	X			X	X	X	X
Visualização de fotos em slideshow	X						X
Drag and drop de ficheiros	X						X
Conversas como árvores	X						X
Controlo de utilizadores (expulsar/convidar participante)	(Brevemente)	X	X	X	X	X	X
Suporte para partilha de aplicações				X		X	
Desktop Sharing				X	X	X	
Whiteboard	X			X		X	X
Suporte de Instant Messaging	X			X	X	X	X
Suporte para serviços de presença	X			X	X	X	X
Gravação e Reprodução de Sessões Síncronas e.g. Áudio / Vídeoconferência				X	X	X	
Controlo de Permissões de Acesso a Funcionalidades	X			X	X	X	X
Agendamento de Sessões				X	X	X	
Suporte para comunicações com utilizadores externos ao sistema	X						X
Protocolos usados	Federation Protocol, baseado em XMPP	HTTP e XML	HTTP	Baseado em T120 com HTTP	HTTP e XML	N.A.	Federation Protocol, baseado em XMPP
Suporte de Telepresença (Qualidade Vídeo SD e HD)	(possível através de gadgets)			X			
Reproduzir histórico de conversações	X						X
Rich Text Editing	X	X	X	X	X	X	X
Persistência	X	X		X		X	X
Dependências	Google Web Toolkit	GWT para modo Offline	Microsoft Word	JVM	JVM	Flash player	N.A.
Disponibiliza API / Suporte para extensão	X	X		X	X	X	X
Suporte para execução em Browser Web	X	X		X		X	X
Acesso Multi-dispositivo	X						X
Segurança	+/- (encriptação de mensagens mas não possui autenticação)	-	-	+	+	+	+

Descrição	Google Wave	Google Docs	CoOffice	Cisco Webex	Microsoft Office Live Communicator	Adobe Connect Pro	Novell Pulse
Escalabilidade	+	-	+/-	+	+/-	+/-	+
Fiabilidade	+	+	+	+	+	+	+

Concluindo, tendo em conta os objectivos da dissertação bem como as contribuições desenvolvidas, da análise anterior resulta que dos serviços estudados se tornou particularmente importante considerar: Google Docs e CoOffice, por serem editores colaborativos de texto rico, que usam aproximações diferentes das usadas neste trabalho para gestão de consistência e controlo de concorrência; Cisco Webex, Microsoft Live Communicator e Adobe Connect Pro por serem serviços colaborativos bastante ricos e por oferecerem diversas funcionalidades, muitas das quais também oferecidas na plataforma base deste trabalho, o PUC, mas usando protocolos diferentes; Novell Pulse, por ter sido desenhado de raiz para ser compatível com o Google Wave e por oferecer em grande parte as mesmas funcionalidades, embora dirigidas para o mundo empresarial. Por fim, foi também de importante a análise feita ao serviço Google Wave por ser, actualmente, a implementação mais avançada dos conceitos, tecnologias e protocolos Wave e por ser a melhor referência existente das funcionalidades que se pretendeu integrar na plataforma desenvolvida.

Os estudos feitos neste capítulo revelaram-se particularmente importantes para todas contribuições desenvolvidas. Para a contribuição da interface portável para o Wave revelou-se importante o estudo do serviço Google Wave. Para a integração do Wave na plataforma revelou-se importante não só o estudo do Google Wave como também do serviço Novell Pulse. Para a contribuição das funcionalidades de *Recording* e *Play* revelou-se muito importante o estudo dos serviços Google Docs e CoOffice, pela parte de importação e exportação de documentos, do serviço Webex, pelas restantes funcionalidades de *Play*, e do Google Wave pelos seus conceitos e tecnologias. Quanto à contribuição do modelo de extensibilidade, revelou-se especialmente interessante o estudo do serviço Google Wave e das suas g*adgets*.

2.4. Comunicações Bidireccionais em Ambientes Web

Comunicações entre clientes e servidores Web são tradicionalmente feitas sobre HTTP. Este protocolo segue um paradigma pedido/resposta em que um cliente liga-se a um servidor, faz um pedido, o servidor envia a resposta e termina a ligação. Este paradigma de comunicação é unidireccional, pois apenas o cliente pode contactar o servidor e este apenas pode comunicar com o cliente em resposta a um pedido. Por outras palavras, segundo o modelo de comunicação tradicional sobre HTTP é impossível para um servidor enviar eventos ou notificações a um cliente sem que este lhe tenha pedido explicitamente. No entanto, é possível simular este tipo de comunicação bidireccional sobre HTTP, recorrendo a diferentes técnicas. Este capítulo vai apresentar e estudar algumas destas técnicas, nomeadamente o protocolo Bayeux [36] e o transporte BOSH [37], assim como um protocolo de transporte alternativo que está actualmente a ser especificado, o protocolo WebSockets [38].

2.4.1. BAYEUX

O Bayeux [36] é um protocolo para comunicações bidireccionais assíncronas entre um cliente e servidor Web. Apesar de permitir outros transportes, o Bayeux foi desenhado tendo em mente o HTTP. As comunicações assíncronas seguem o modelo *publish*/*subscribe*, através do uso de *channels* (equivalente a tópicos na visão tradicional do modelo). Usando os canais é possível enviar mensagens do cliente para o servidor, do servidor para o cliente e entre dois clientes diferentes, usando o servidor como *router*.

O envio assíncrono de mensagens do servidor para o cliente Web é geralmente conhecido como *server-push*. A combinação de diferentes técnicas de *server-push* com aplicações Ajax é também conhecida como *Comet* [36]. O objectivo por traz da criação do protocolo Bayeux é unificar as diferentes implementações e Frameworks *Comet* actualmente existentes sobre uma única especificação generalizada e normalizada. Apesar de [36] não ser um RFC oficial, o documento segue mesmo estilo de escrita.

Dadas as limitações expostas no inicio deste capítulo a cerca do envio de eventos sobre HTTP de um servidor para um cliente Web, a solução do Bayeux é usar múltiplas ligações HTTP. A ideia é que haja sempre uma ligação aberta para que o servidor possa enviar eventos assincronamente. Para tal, o cliente faz um pedido ao servidor e este não lhe envia logo a resposta, deixando assim a ligação suspensa.

Existe, no entanto, uma limitação a esta aproximação. Esta está no número de ligações que um cliente pode manter simultaneamente abertas com um servidor. Segundo [39], um cliente Web não deve manter mais do que duas ligações com qualquer servidor, sendo esta política implementada pela maior parte dos browsers Web. Como tal, o Bayeux nunca exige mais do que duas ligações simultâneas entre um cliente e um servidor. Mais especificamente, o que o Bayeux

faz é manter duas ligações sempre abertas, de modo a permitir comunicação bidireccional assíncrona.

Por fim, falta referir que existem diferentes maneiras de gerir as ligações entre clientes e servidores e de efectuar o transporte das comunicações. Destas destacam-se *Polling* e *Streaming*. *Polling* tradicional consiste em ter o cliente a efectuar pedidos constantes e frequentes ao servidor para receber eventos, o que é bastante dispendioso. Duas variantes, implementadas pelo Bayeux, são *Long-Polling* e *Callback-Polling*. *Long-Polling* consiste em manter a ligação criada pelo pedido do cliente aberta até haver eventos para lhe enviar. Como descrito anteriormente esta é a forma estandardizada de operação no Bayeux. *Callback-Polling* é uma pequena variante do anterior, onde as respostas do servidor são enviadas num Javascript *callback wrapper*, de forma a facilitar a sua entrega. O que as diferentes formas de *Polling* têm em comum é que depois de enviar a resposta, o servidor termina sempre a ligação e exige que o cliente abra uma nova ligação. Por outro lado, técnicas de *Streaming* tentam sempre manter a mesma ligação com o cliente aberta, de forma a optimizar latência e mensagens extra resultantes de abrir várias ligações. Porém, estas formas de transporte são menos portáveis pois estão limitadas pelas capacidades dos clientes e por necessitarem de proxies de forma a poderem enviar respostas HTTP incompletas. Transportes de *Streaming* são opcionais no Bayeux, ao contrário dos transportes *Long-Polling* e *Callback Polling* que são obrigatórios em qualquer implementação do protocolo.

2.4.2. BOSH

O BOSH [37] é um protocolo de transporte que define como emular uma ligação TCP, *long-lived* e bidireccional, usando apenas HTTP. O método usado no BOSH consiste em usar múltiplas ligações HTTP síncronas. Mais especificamente, o BOSH pode usar simultaneamente até duas ligações HTTP síncronas para emular a comunicação bidireccional. O cliente tenta sempre manter uma ligação aberta, para que o servidor lhe possa enviar eventos. Se quiser enviar mais dados para o servidor, o cliente usa a segunda ligação. O servidor, ao receber o segundo pedido fecha a ligação que já estava aberta, mesmo que não tenha dados para enviar ao cliente, de modo a que se este necessitar de enviar mais dados o possa fazer.

Em [37] é feito um esforço no sentido de afastar o BOSH das tecnologias *Comet* e de afirmar a sua superioridade face a estas. No entanto o BOSH não é mais que uma implementação de uma dessas tecnologias, o *Long-Polling*. Quanto à referida superioridade, esta é relativa a apenas outras tecnologias *Comet*, como *Streaming* e *Hidden IFrame*. Por outras palavras o BOSH está bastante relacionado com o Bayeux, enquanto protocolo de comunicações do tipo *Comet*.

2.4.3. Websockets

O protocolo WebSocket [38] é um novo protocolo que está a ser especificado pelo IETF, em conjunto com o W3C [40]. O objectivo deste protocolo é substituir o uso do protocolo HTTP para comunicações bidireccionais na Web, dadas as limitações do HTTP para esse tipo de comunicação e dados os elevados custos das tecnologias *Comet*. O protocolo WebSocket é um protocolo de nível aplicacional, pensado para ser usado em conjunto com o protocolo de encaminhamento TCP. Como tal, o WebSocket tenta expor às aplicações a semântica pura do TCP, adicionando-lhe apenas o necessário para superar as limitações da Web. O protocolo está também a ser desenhado de forma a ser compatível com o HTTP, partilhando a mesma porta que este e fazendo do seu *handshake* um HTTP *Upgrade Handshake* válido. Assim torna-se bastante simples evoluir de uma ligação HTTP simples para uma ligação WebSocket.

O protocolo WebSocket traz grandes vantagens face a tecnologias *Comet*, em termos de redução de tráfego Web desnecessário, redução de latência e, consequentemente, aumento de escalabilidade. No entanto o protocolo ainda tem alguns problemas. Para além de ser bastante novo e a sua utilização real e suporte nos grandes browsers ainda ser bastante reduzida, o protocolo apresenta problemas em redes com Proxies HTTP intermédios. Estes problemas devem-se, por um lado, por alguns Proxies removerem os cabeçalhos dos pedidos dirigidos aos servidores e, por outro lado, por os Proxies ainda não serem capazes de processar tráfego do protocolo WebSocket. Porém espera-se que no futuro comecem a ser produzidos Proxies capazes de processar ambos os tipos de tráfego e que este protocolo seja adoptado em larga escala.

2.4.4. Sumário

Neste capítulo foram analisadas diferentes tecnologias para conseguir comunicações bidireccionais em ambientes Web. Dado o protocolo aplicacional predominante na Web ser o HTTP, o foco principal da análise feita incidiu em técnicas para superar as limitações desse protocolo para este tipo de comunicações. Da análise feita destaca-se o Bayeux [36] por ser um protocolo comunicacional especificado exactamente para este tipo de comunicações e por ser compatível com diferentes protocolos de transporte e diferentes técnicas. Destaca-se ainda o BOSH [37], como um bom exemplo de um protocolo de transporte sobre HTTP que usa a técnica *Long-Polling* para simular comunicações bidireccionais. Por fim, foi também analisado neste capítulo um protocolo aplicacional alternativo ao HTTP, o protocolo de WebSockets [38]. Apesar de ainda estar nas primeiras fases de especificação, este protocolo promete superar as limitações do HTTP para comunicações bidireccionais em ambientes Web, oferecendo ao mesmo tempo melhor performance do que as tecnologias apresentadas em [36] e [37].

O estudo feito neste capítulo revelou-se particularmente importante para a concepção e implementação da primeira contribuição desta dissertação, a interface portável para o Wave.

3. Concepção

Pretendeu-se desenvolver uma plataforma extensível que facilitasse a criação de aplicações colaborativas, oferecendo diversos recursos colaborativos sob uma interface unificada. Para tal, partiu-se da plataforma PUC (cap. 2.2.5) e expandiu-se as suas funcionalidades, criando uma plataforma colaborativa ainda mais rica em termos de experiência de utilização para os utilizadores finais.

Este capítulo irá apresentar uma visão conceptual e arquitectural das contribuições desenvolvidas nesta dissertação. O capítulo está dividido em quatro subcapítulos, correspondendo cada um a uma contribuição diferente.

3.1. Interface Portável para o Wave

A plataforma desenvolvida tem como objectivo fornecer serviços colaborativos a dispositivos heterogéneos. É portanto crucial garantir a portabilidade de todos os seus componentes.

Ao integrar o Wave na plataforma, surge a necessidade de criar um meio de comunicação entre este e os dispositivos heterogéneos, de modo a que possam ser efectuadas operações computacionalmente mais pesadas. Normalmente tal procedimento poderia ser feito através dos protocolos cliente-servidor usados pelos recursos. Porém no caso do Wave, devido à falta de portabilidade do seu protocolo, baseado em invocações RPC e XML sobre TCP, é necessário conceber uma interface portável para o mesmo.

De modo a superar este problema foi concebida uma solução pensada de raiz para ser o mais portável possível, ou seja, que pudesse ser utilizada no maior número de dispositivos heterogéneos diferentes.

Um dos modos de comunicação mais portável é através da Web e do protocolo HTTP. Quase todos os dispositivos móveis actuais possuem *browsers* Web e suportam esse tipo de comunicações. Assim foi decidido que o modelo a adoptar para esta contribuição deveria ser um modelo Web/HTTP. No entanto este modelo apresenta um grave problema face às necessidades do Wave. Invocações Web seguem um paradigma de pedido resposta, onde o servidor Web apenas pode enviar dados ao cliente quando este lhe pede. O Wave, de forma a permitir

colaboração em tempo quase real, necessita de um protocolo de comunicação bidireccional com bom tempo de resposta.

Por outras palavras, a interface a conceber deveria ser não só o mais portável possível mas deveria também permitir comunicações bidireccionais. Uma vez que a portabilidade oferecida pelo modelo Web/HTTP era insubstituível, procurou-se uma solução bidireccional sobre HTTP. Esta solução surgiu sobre a forma do protocolo Bayeux [36]. No capítulo 2.4.1 foi feita uma análise extensiva do protocolo. Resumindo essa análise, o Bayeux é um protocolo que permite comunicações bidireccionais e assíncronas sobre HTTP. Com a sua utilização era possível resolver ambos os problemas descritos.

Para integrar esta interface portável no Wave havia duas soluções possíveis. A primeira consistia em alterar o código interno do servidor Wave e substituir completamente o seu protocolo cliente-servidor pela nova interface. Esta aproximação, para além de tornar a interface bastante dependente dos detalhes do servidor Wave e dificultar o processo de actualização do mesmo, obrigava à obtenção de um elevado e detalhado nível de conhecimento do seu código interno. A outra solução, que permitia maior abstracção e independência do servidor Wave, consistia em construir um componente intermédio que fizesse a ligação entre os clientes e o servidor Wave. Este componente, que funcionaria como uma *proxy* ou *gateway*, comunicaria com o servidor Wave através do seu protocolo cliente-servidor e iria expor para os clientes a interface portável concebida. Depois de uma extensa análise das vantagens e desvantagens das duas aproximações, optou-se pela última. A figura 5 apresenta a arquitectura da solução concebida.

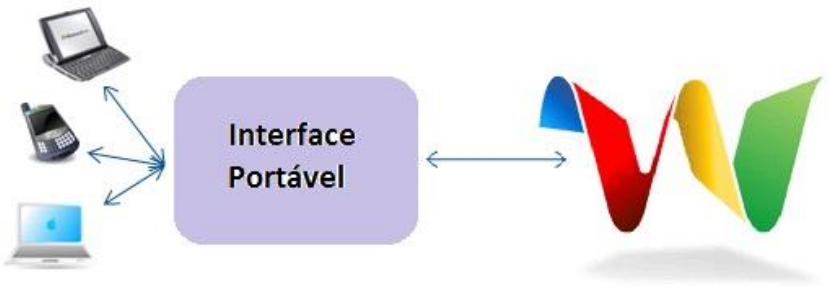


Figura 5 – Arquitectura da Interface Portável para o Wave

3.2. Integração do Wave na Plataforma

A plataforma desenvolvida oferece diversas funcionalidades colaborativas às suas aplicações cliente. De forma a disponibilizar estes serviços, a plataforma recorre a diferentes servidores colaborativos, cada um com o seu conjunto de funcionalidades distintas.

Uma das mais importantes contribuições esperadas para esta dissertação era a inclusão de funcionalidades de edição colaborativa de documentos na plataforma. A solução concebida para esta contribuição consistiu em adicionar um novo servidor de recursos à plataforma e desenvolver todo o suporte necessário, do lado desta, para este novo recurso.

O servidor de recursos integrado na plataforma foi um servidor Wave. O Wave, através de controlo de concorrência optimístico, permite edição colaborativa de documentos entre vários clientes e com pouca latência. Um estudo detalhado do Wave pode ser encontrado no capítulo 2.3.5. Para integrar o Wave não era necessária qualquer alteração no seu servidor. No entanto, era inevitável desenvolver alguns componentes do lado da plataforma, assim como alterar outros já existentes.

Como a plataforma que lhe serve de base, cuja análise detalhada é feita no capítulo 2.2.5, a plataforma desenvolvida está logicamente dividida em duas camadas principais: a camada central, ou *Core*, que contém toda a lógica do agendamento e gestão das sessões colaborativas e a camada dos recursos, *Resource*, onde são geridos os recursos e é feita a comunicação com os seus servidores. Fruto desta separação lógica, para integrar um novo recurso na plataforma a maior parte do trabalho a realizar teria que ser feito na camada dos recursos. Salvo algumas alterações pontuais, inerentes de pequenos problemas de modularidade herdados da plataforma base, a camada central mantinha-se completamente inalterada e abstraída do novo recurso.

Para integrar o Wave na plataforma foi necessário conceber dois novos componentes na camada *Resource*. O primeiro componente, denominado de *WaveConnector*, trata de toda a comunicação entre o servidor Wave e a plataforma. Mais especificamente, este componente é responsável tanto por enviar operações pedidas pela plataforma para executar no servidor Wave, como de reencaminhar eventos disparados no servidor Wave para serem processados na plataforma.

De forma a poder-se abstrair o *Core* dos detalhes do servidor Wave é necessário conceber outro componente na camada *Resource*. Este segundo componente, denominado de *WaveController*, tem como responsabilidade receber os pedidos colaborativos genéricos feitos pela camada *Core* e mapeá-los em operações concretas no servidor Wave, assim como processar os eventos recebidos do Wave e convertê-los em eventos colaborativos genéricos que possam ser interpretados pelo *Core*. Em termos conceptuais, o *WaveController* representa uma instância de Wave no contexto de uma sessão colaborativa. Para fazer esta ligação entre a camada superior e o servidor de recursos, o *WaveController* comunica internamente com o *WaveConnector* e com o componente do *Core* que representa a sessão, o *SessionManager*. O *WaveController* é também responsável por comunicar com a base de dados da plataforma para persistir todos os dados relativos à sessão de recursos que representa.

Como foi dito anteriormente, era inevitável fazer algumas alterações no *Core* para suportar o novo tipo de recurso. No entanto, estas alterações incidiram apenas no componente

SessionManager e no componente do *Core* responsável por gerir todos os recursos, o *ResourceManager*. A figura 6 representa a arquitectura da solução concebida.

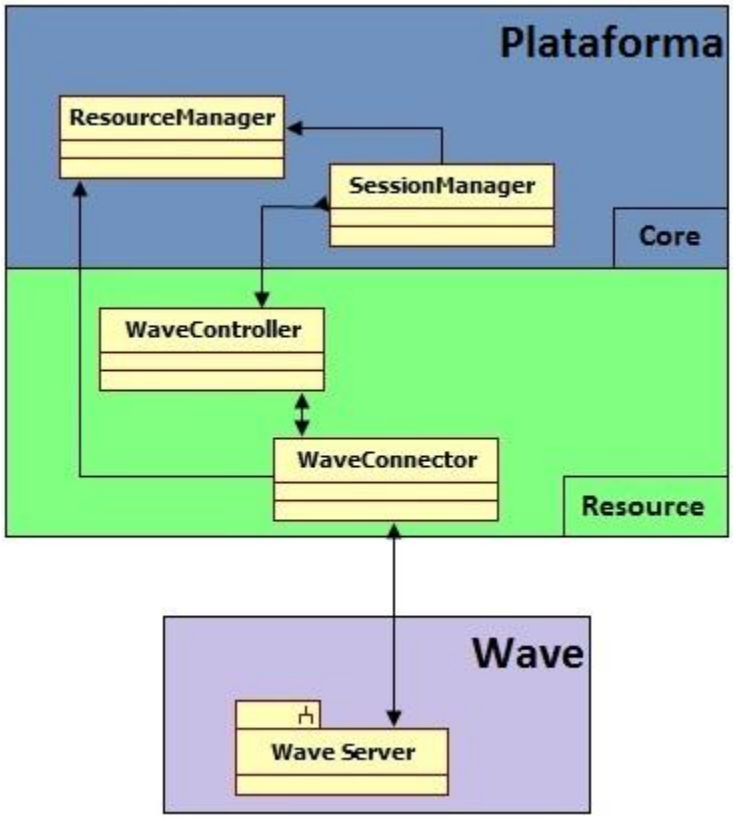


Figura 6 - Arquitectura da solução concebida

3.3. Funcionalidades de *Recoding* e *Play*

O grande tema e objectivo desta dissertação foi a introdução dos conceitos, tecnologias e funcionalidades apresentadas pelo Google Wave na plataforma colaborativa a desenvolver. Como vimos no capítulo anterior, uma das contribuições derivadas deste objectivo foi a inclusão do Wave como um novo servidor de recursos da plataforma, adicionando-lhe assim funcionalidades de edição colaborativa de documentos. Porém, os conceitos apresentados pelo Wave não se limitam apenas às funcionalidades obtidas através da sua integração como servidor de recursos. Outros conceitos ou funcionalidades secundárias do Wave têm elevado potencial de usabilidade e interactividade. Como tal, podem ser adaptados e integrados nas camadas da plataforma, enriquecendo ainda mais a experiência da sua utilização para as aplicações cliente e os seus utilizadores.

Exemplo destes conceitos é a funcionalidade de reprodução do histórico de uma Wavelet. Através desta funcionalidade é possível, para um utilizador que tenha sido adicionado a meio de

uma conversa, ver todo o fluxo colaborativo que decorreu desde a criação da mesma até à sua entrada. Outro exemplo de uso desta funcionalidade é a exportação de uma sessão de edição colaborativa de um documento para utilizadores exteriores a esta, podendo estes seguir todo o seu fluxo como se tratasse de uma apresentação.

Outro conceito interessante provém da unidade mais básica do Wave, o Blip. Sendo uma Wave (conversa) composta por várias Wavelets (sessões) e estando cada uma destas dividida em vários Blips, um Blip acaba por ser um excerto de uma sessão ou conversa e pode conter desde uma simples mensagem a um documento complexo. Apesar da sua simplicidade, esta organização lógica do modelo de dados do Wave permite, por um lado, organizar uma conversa segundo diversos critérios e, por outro lado, usar funcionalidades como a de reprodução do histórico de uma Wavelet, como vimos no parágrafo anterior.

Com esta contribuição pretendeu-se juntar estes dois conceitos e mapeá-los numa nova funcionalidade a adicionar ao *Core* da plataforma. A esta funcionalidade foi dada o nome de *Recording* e *Play*: *Recording* pela introdução do conceito de *Blip* no modelo de dados e *Play* pela reprodução do histórico de uma sessão. O objectivo era permitir aos clientes da plataforma definir um processo que grava-se e separa-se o conteúdo de uma sessão colaborativa em pequenos excertos, delimitados segundo diversos critérios. Posteriormente estes excertos poderiam ser interpretados pelos clientes, possibilitando assim a reprodução da sessão.

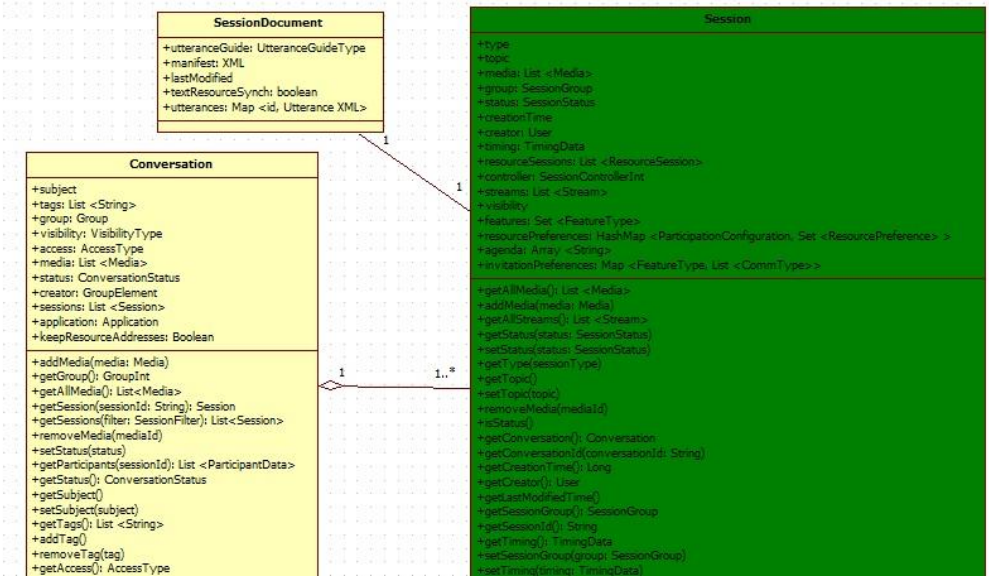


Figura 7 – Pequena parte do modelo de dados da plataforma com o módulo concebido

Esta contribuição dividiu-se em duas fases separadas. Em primeiro lugar era fundamental conceber o suporte para a separação e armazenamento dos excertos de uma sessão. Para tal, foi incorporado no modelo de dados da plataforma um novo módulo. A figura 7 representa uma pequena parte do modelo de dados da plataforma. Na figura são visíveis os componentes *Session* (sessão) e *Conversation* (conversa). O outro componente, *SessionDocument*, representa o

módulo concebido, isto é, representa todos os excertos de uma sessão. Este componente está directamente ligado à sessão numa relação de um para um. Assim, foi discutido se este componente deveria realmente existir ou se a própria sessão deveria assumir as suas funções. No entanto, como são conceitos distintos e têm funções e responsabilidades separadas, as boas práticas de modularidade exigiam que o novo componente fosse desenvolvido em separado da sessão.

Como foi dito, o *SessionDocument* é composto pelos vários excertos da sessão, denominados no modelo de dados por *Utterances*. Cada *Utterance* é um documento XML com toda a estrutura do excerto de sessão que representa: quais os recursos colaborativos que foram utilizados na sessão, como se relacionam entre si e como podem ser obtidos (geralmente através de um URL). Para além de cada *Utterance* expor a estrutura do excerto que representa, o *SessionDocument* possui um documento XML único, chamado *Manifest*, que apresenta a estrutura desse *SessionDocument*, ou seja, como os vários excertos da sessão estão relacionados entre si. Este conceito de usar documentos XML para representar os excertos e usar um segundo documento com a sua estrutura provém também do Wave, onde os vários Blips são documentos XML e existe um tipo especial de Blip que contém a estrutura da Wavelet e diz como os Blips se relacionam entre si.

Por fim, o *SessionDocument* é composto por mais dois atributos. O primeiro atributo é um booleano que define se deve haver sincronização com recursos de texto. Se este atributo for verdadeiro, sempre que é fechada uma *Utterance* e antes de ser iniciada a seguinte o seu documento XML é convertido e submetido aos recursos de texto que estejam a ser utilizados na sessão, de modo a que os participantes possam seguir o processo de *Recording* em tempo real. O segundo atributo é o *UtteranceGuide*. Este atributo representa o critério de delimitação das *Utterances*, quando acaba uma e começa outra. Foram concebidos diferentes casos de uso, incluindo: *Utterances* guiadas pela transição de slides, ou seja, numa sessão cujo recurso colaborativo principal é uma apresentação (SlideShow), cada vez que a apresentação progride e é mudado o slide é criada uma nova *Utterance*; *Utterances* guiadas por silêncio de voz numa sessão cujo recurso principal é uma apresentação de áudio; *Utterances* guiadas por parágrafos numa sessão colaborativa de edição de documentos; *Utterances* guiadas pela autorização de fala (*Speaker Role*), novamente numa apresentação de áudio; etc.

De modo a possibilitar estes casos de uso era imprescindível saber a altura em que se davam alterações nos recursos: quando havia uma transição de slide, silêncio de voz, etc. Para tal, pensou-se que a melhor solução seria o módulo concebido receber estes eventos quando eram enviados para as aplicações cliente. Porém, apesar de a plataforma base possuir toda a lógica para o processamento dos eventos, esta ainda não tinha um mecanismo que lhe permitisse envia-los para os clientes. Era portanto necessário primeiro conceber um sistema que fizesse este envio. Depois, o sistema concebido seria usado pelo módulo anterior para despoletar a criação de novas *Utterances.*

Como o lançamento de eventos e a sua recepção são procedimentos de natureza assíncrona, a solução concebida também deveria ser um processo assíncrono. Decidiu-se assim usar um MOM (Message Oriented Middleware) para enviar os eventos para as aplicações. Usando este sistema os clientes, quando quisessem começar a receber os eventos de uma sessão, poderiam definir assincronamente esse interesse junto do sistema, sem ter que contactar a plataforma. Por outro lado, a plataforma transformava os eventos em mensagens e enviava-as para o MOM, sem ter que saber quantos ou quais os clientes interessados.

Depois de concebido o sistema de envio de eventos para as aplicações, definiu-se o módulo de *Recording* e *Play* para também os receber. Mais uma vez prova-se a vantagem da escolha de um sistema assíncrono, pois com este o módulo apenas necessita de registar nele o seu interesse e o resto da plataforma pode permanecer inalterada. A figura 8 mostra a arquitectura da solução concebida, com o sistema de MOM, a plataforma, o módulo de *Recording* e *Play* e as aplicações cliente da plataforma.

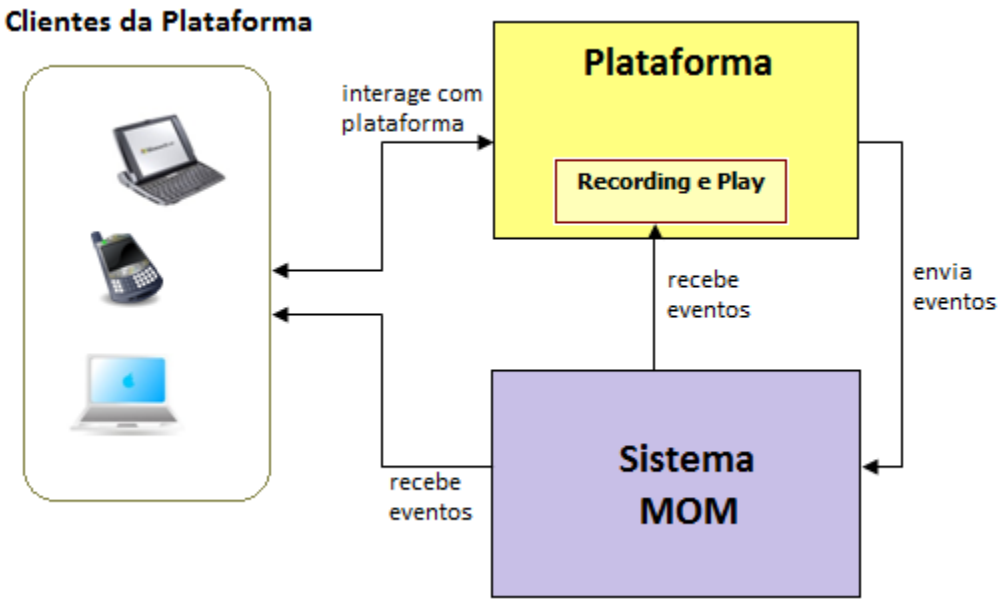


Figura 8 – Sistema MOM e interacções entre este, a plataforma e os seus clientes

Concebido o suporte e terminada a primeira fase de *Recording*, faltava fazer uso do mesmo na segunda fase de *Play*. Seguindo o exemplo do Wave, a reprodução do histórico de uma Wavelet é um processo de certa forma simplificado pois apenas envolve documentos de texto. No caso da plataforma, como são usados diversos tipos de recursos colaborativos o processo torna-se mais complexo, pois a reprodução de cada recurso é feita de maneira diferente. Por exemplo, no caso de recursos de áudio/vídeo era necessário um mecanismo que permitisse iniciar e terminar a sua gravação e posteriormente obtê-la remotamente ou fazer *streaming* da mesma; no caso de apresentações de slides teria que ser possível obter cada um dos slides em separado; etc. A concepção e implementação destes mecanismos para cada tipo de recurso existente na plataforma ia para além do escopo desta dissertação. No entanto, uma vez que o grande foco

deste trabalho incidiu sobre o Wave, era imprescindível desenvolver os mecanismos para este recurso.

No caso do Wave, só com a utilização das *Utterances* era possível saber os documentos (Blips) que tivessem sido criados ou editados na duração do excerto de uma sessão. No entanto, para visualizar esses documentos era obrigatório usar o servidor Wave. Não havia uma maneira de visualizar ou editar fora do Wave um documento colaborativo criado neste. No contexto desta contribuição tornava-se portanto essencial conceber um módulo para exportação de documentos Wave. Uma vez exportados, os documentos poderiam ser editados e, como tal, era também necessário conceber um módulo para importação de documentos para o Wave.

Como existe actualmente uma grande variedade de formatos para documentos de texto rico, a solução concebida consistiu em escolher um formato de vasta utilização e suficientemente versátil para conter todas as anotações e meta-dados provenientes do Wave. Deveria ainda ser possível converter facilmente deste formato para outros. Assim bastaria desenvolver um módulo que fizesse a conversão de Wave para este formato e vice-versa, usando-se depois ferramentas *open source* para converter entre este formato e outros.

O formato escolhido foi o HTML. Juntamente com CSS, o HTML possibilitava a preservação de todo o texto e anotações existentes no Wave. Como é um formato vastamente utilizado, existem diversas ferramentas para conversão entre ele e outros formatos. Com este objectivo em mente, foram concebidos dois novos módulos, um para importação de documentos (conversão de HTML para Wave) e outro para exportação de documentos (conversão de Wave para HTML). Como estas funcionalidades são de grande interesse não só para a plataforma como para qualquer aplicação que use o Wave, decidiu-se incorporar estes módulos numa biblioteca à parte da plataforma e do servidor Wave. Chegou-se a considerar incorporar os métodos directamente no servidor Wave, no entanto tal aproximação aumentaria a dependência entre a plataforma e o mesmo e tornaria a sua actualização mais difícil. A biblioteca criada possuía dois métodos: um para conversão de Blips para HTML e outro para o processo inverso. Do lado da plataforma, foi criado o suporte na camada *Resource* para invocar estas funcionalidades e obter ou enviar os documentos convertidos. Para além de Blips singulares, foi também adicionado à plataforma o suporte para a exportação de Waves inteiras, assim como a importação automática de vários documentos para diferentes Blips de uma Wavelet singular. Por fim, desenvolveu-se soluções para a conversão entre HTML e outros formatos, incluindo DOCX e PDF.

3.4. Modelo de Extensibilidade

No seguimento do trabalho feito no capítulo 3.3, continuou-se a analisar os conceitos e funcionalidades secundárias do Wave. Depois dos que já tinham sido aproveitados nas contribuições anteriores, restava um que demonstrava elevado potencial colaborativo, interactivo

e de usabilidade. Este era o conceito de *Gadget* (ver cap. 2.3.5). Através desta funcionalidade foi possível para os criadores do Wave definir um modelo de extensibilidade praticamente ilimitado, que tirava partido do poder da comunidade e da criatividade de terceiros. Como tal, havia grande interesse em adoptar este conceito na plataforma.

O conceito de *Gadget* no Wave está relacionado com um tipo de aplicação mais conhecido e vastamente utilizado na Web, denominado de *Widget*. Segundo [42], *Widgets* são aplicações que correm nos clientes ou são embebidas em documentos Web e que são criadas usando normas Web. Geralmente *Widgets* são também conhecidas por serem aplicações leves com um objectivo bastante simples e concreto, sendo criadas por terceiros e depois embebidas em aplicações Web mais complexas, como o Wave. Como cada *Widget* pode ter propósitos e naturezas distintas, para poderem ser utilizadas de forma uniforme as *Widgets* são geralmente processadas por uma *Widget Engine*. Para tal seguem alguma norma Web especificada por esta. O papel da *Widget Engine* é processar diferentes Widgets que sigam a sua norma e devolver um método de acesso portável ao conteúdo das mesmas.

Face ao conceito tradicional de *Widget*, o que as *Gadgets* do Wave trazem de novo é a colaboração. A maior parte das *Widgets* existentes antes do Wave foram criadas com o objectivo de enriquecer a experiência de utilização de um cliente singular. No entanto, como o grande foco do Wave é a colaboração entre vários clientes, também as *Widgets* do Wave (*Gadgets*) têm um cariz colaborativo e podem ser usadas simultaneamente entre os vários participantes da Wave onde são inseridas. Exemplos de *Gadgets* vão desde aplicações simples como jogos a aplicações mais complexas como conferências Web.

Com esta contribuição pretendeu-se desenvolver um modelo de extensibilidade baseado no modelo de *Gadgets* do Wave. Para tal era necessário, por um lado, escolher uma especificação que permitisse criar *Widgets* colaborativas como as *Gadegts* do Wave e, por outro lado, conceber uma forma de processar estas *Widgets* e devolver um acesso portável ao seu conteúdo. Diferentes especificações de *Widgets* foram analisadas, incluindo OpenSocial *Widgets* [43] e W3C *Widgets* [42]. No entanto, a única especificação encontrada de natureza colaborativa é a das *Gadgets* do Wave [44]. Portanto, iria ser necessário conceber uma forma de processar *Gadgets* do Wave na plataforma. Tal seria feito através de um *Widget Egine* para *Gadgets* Wave.

Escolhido o *Widget Engine*, era necessário integrá-lo na plataforma. Esta integração era possível de duas formas. A primeira solução pensada consistia em adicionar as funcionalidades do *Widget Engine* directamente na camada *Core* da plataforma. Esta aproximação tinha como vantagens tornar as *Widgets* colaborativas numa funcionalidade nativa da plataforma, disponível para todas as sessões, independentemente dos recursos que estivessem a utilizar. No entanto, esta aproximação ia contra a própria especificação da plataforma, pois tornava o *Core* dependente de um serviço externo e tirava-lhe algumas das suas propriedades de abstracção. A outra solução consistia em ver o *Widget Engine* como mais um servidor de recursos e adicionar o suporte para a sua utilização na camada *Resource*. Esta aproximação conformava com a especificação da

plataforma e permitia, tal como a primeira aproximação pensada, que qualquer sessão contivesse *Widgets* colaborativos, pois, segundo a especificação da plataforma (ver capítulo 2.3.5), uma sessão pode conter vários recursos simultaneamente.

Uma vez que a solução concebida para esta contribuição passou a consistir em adicionar um novo servidor de recursos à plataforma, a mesma ficou muito semelhante à solução concebida no capítulo 3.2. Tal como foi descrito nesse capítulo, era necessário adicionar à camada *Resource* da plataforma dois novos componentes: *WidgetConnector* e *WidgetController*. O nome dos componentes seria depois alterado para reflectir o *Widget Engine* escolhido. O primeiro componente descrito ficaria responsável por comunicar com o *Widget Engine*, enviando-lhe pedidos e recebendo eventos. O segundo componente ficaria responsável por controlar todos os detalhes da sessão de recursos que representava. Para tal, comunicaria com o primeiro componente e com o componente da camada *Core* que representa a sessão, o *SessionManager*. Como no capítulo 3.2, também houve a necessidade de efectuar pequenas alterações neste componente para suportar o novo tipo de recurso, assim como no *ResourceManager*. A figura 9 apresenta a arquitectura da solução concebida.

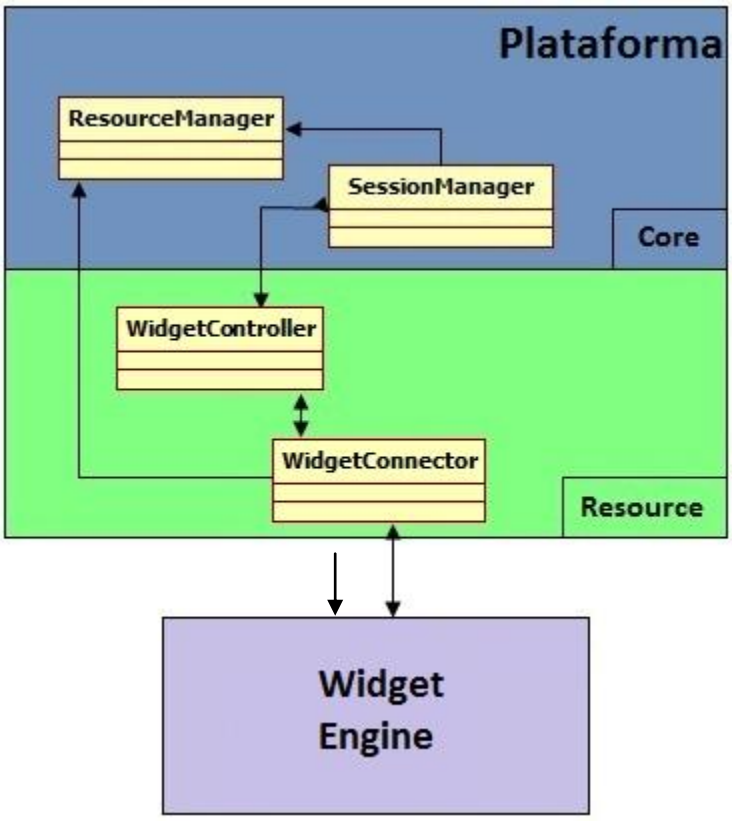


Figura 9 – Arquitectura da solução concebida

No entanto havia um pormenor que nesta solução diferia da apresentada no capítulo 3.2. Como o *Widget Engine* pode processar um número ilimitado de *Widgets* diferentes, existe a necessidade, aquando da criação e *setup* de uma sessão na plataforma, de indicar qual a *Widget*

que se pretende usar. Esta necessidade não existia no Wave porque todas as Waves, quando criadas, são iguais. Para resolver este problema foi concebido um sistema de carregamento dinâmico de tipos de recursos, substituindo assim o sistema estático da plataforma base. A plataforma base usava um sistema estático baseado em enumerações. Este sistema, apesar de eficiente, era pouco extensível e qualquer adição de um novo recurso implicava a sua alteração. O sistema extensível concebido baseou-se na adição de uma nova entidade ao modelo de dados da plataforma. Esta entidade, denominada de *ResourceFeature*, representava um recurso que poderia ser usado numa sessão da plataforma. A entidade foi definida como contendo um nome, uma descrição e um url para o seu ícone, se o possuísse. Assim, as aplicações cliente em vez de só poderem saber estaticamente em *compile time* quais os recursos disponíveis, passavam a poder determinar esta informação dinamicamente em *runtime time*. Para tal, foi também adicionado um novo método ao componente *ResourceManager* que iria consultar na base de dados da plataforma quais os *ResourceFeatures* disponíveis e devolvê-los-ia. Por fim, foi concebido um sistema que periodicamente comunicaria com os servidores de recursos de forma a actualizar a base de dados com novos *ResourceFeatures*. Este sistema tinha particular interesse no contexto desta contribuição e da adição de *Widgets* à plataforma pois assim a plataforma poderia actualizar-se automaticamente quando fossem adicionadas novas *Widgets* à *Widget Engine*. A figura 10 apresenta um pequeno excerto do modelo de dados da plataforma com as novas alterações.

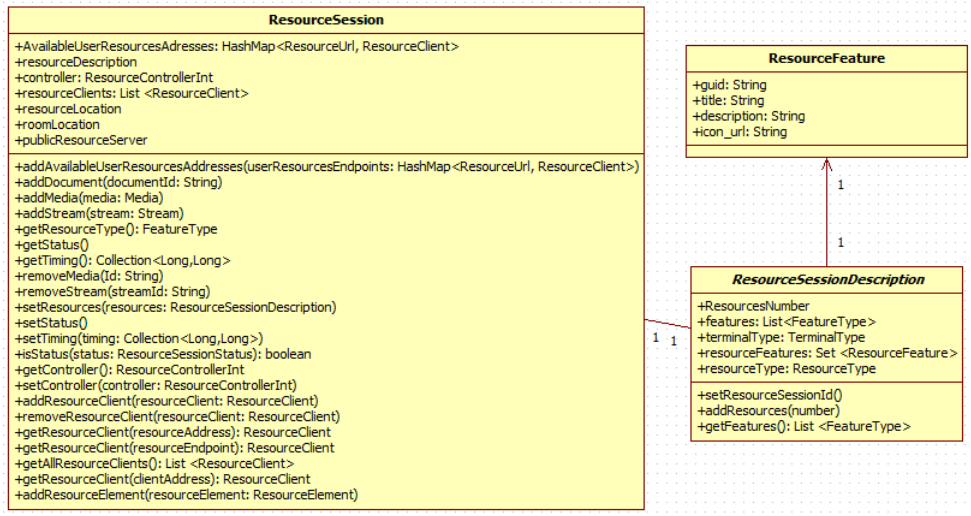


Figura 10 – Pequena parte do modelo de dados com o componente ResourceFeature

4. Implementação

Este capítulo pretende demonstrar em detalhe como foram implementadas cada uma das contribuições desta dissertação e que tecnologias específicas foram usadas para a realização das soluções concebidas no capítulo 3.

4.1. Interface Portável para o Wave

No capítulo 3.1 foi exposta a solução concebida para esta contribuição. Resumindo, a solução apresentada consistia em desenvolver um módulo que faria a ponte entre os clientes da plataforma e o servidor Wave. A comunicação entre os clientes e o módulo seria feita através do protocolo Bayeux sobre HTTP e a comunicação entre o módulo e o servidor Wave seria feita através do protocolo cliente-servidor do Wave.

4.1.1. O Servidor Wave

O primeiro ponto a abordar nesta contribuição era o servidor Wave. Para usar as funcionalidades e tecnologias do Wave era necessário um servidor Wave funcional, que implementasse os protocolos e modelos especificados pela Google (ver capítulos 2.1.6, 2.2.4 e 2.3.5). Este ponto era de extrema importância, pois acabava por afectar directa ou indirectamente não só esta mas todas as contribuições da dissertação.

Como se pode deduzir pela extensa análise feita ao Wave nos capítulos 2.1.6, 2.2.4 e 2.3.5, implementar de raiz um servidor Wave era uma tarefa bastante complexa e saia fora do escopo desta dissertação. Era portanto necessário escolher um servidor Wave já existente. As escolhas possíveis agrupavam-se sobre serviços proprietários e servidores *open source*. A primeira escolha não era viável pois comprometia a privacidade e segurança dos clientes da plataforma, obrigava a que estes se registassem nesses serviços, não permitia controlar o funcionamento do servidor e estava dependente da existência de alguma forma de API. Quanto aos servidores *open source*, como o Wave é um conceito relativamente novo ainda existiam poucas implementações. Para além do PyGoWave [70] e Ruby on Sails [56], que são servidores ainda muito básicos e pouco desenvolvidos, o único servidor Wave *open source* capaz de oferecer as funcionalidades necessárias para este trabalho era o servidor de referência da Google, o FedOne [4]. Para além de

ser o servidor *open source* mais avançado até à data de escrita desta dissertação, o FedOne contava com uma forte comunidade e era gerido pela própria Google. Como tal, foi escolhido para ser o servidor Wave a usar nas diferentes contribuições desta dissertação.

Como no simples *console client* que vinha incluído no seu projecto, para comunicar com os seus clientes o FedOne usava um protocolo RPC baseado em XML sobre TCP (Protocol Buffers [6]). Como foi dito no capítulo 3.1, este protocolo era pouco portável e daí a necessidade de conceber e implementar esta contribuição. No entanto, esta era a única forma de comunicar com o FedOne, ou seja, teria que ser usada pelo módulo concebido para fazer a ponte entre os clientes da plataforma e o servidor.

Como tanto o FedOne como a plataforma estavam escritos na linguagem Java, aproveitou-se algum do código usado no FedOne e no seu *console client* para facilitar a comunicação com o servidor. Mais especificamente o FedOne possuía uma abstracção, chamada de *ClientBackend*, que facilitava a interacção com o servidor e abstraía dos detalhes do protocolo RPC. Esta abstracção era usada directamente pelo *console client* e por uma outra abstracção de mais alto nível, a *Agent* API. Esta API era usada para criar agentes que se ligavam às Waves e executavam tarefas específicas. A figura 11, retirada da documentação do FedOne, demonstra o exemplo de três agentes existentes no serviço Google Wave Preview [5].

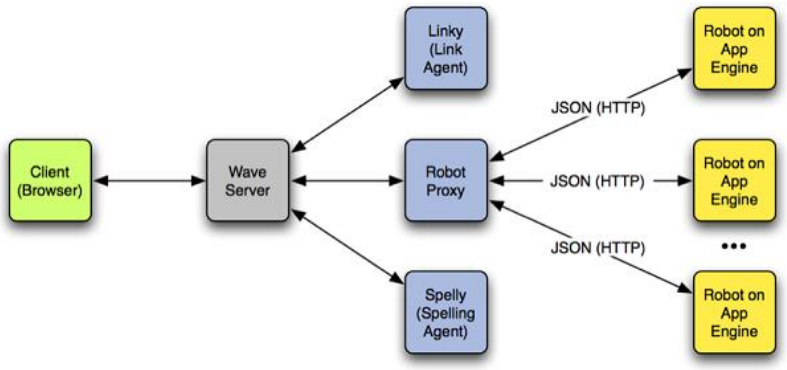


Figura 11 – Exemplo de uso de agentes no serviço Google Wave

O uso desta API seria a forma normalizada de integrar o servidor Wave noutras plataformas. No entanto, no FedOne esta API era apenas um *wrapper* à volta da abstracção *ClientBackend* que apesar de abstrair diversas operações e tornar o processo de integração mais fácil, também limitava a autonomia do agente e as notificações que este podia receber do servidor. Em particular, a *Agent* API impedia os seus agentes de receber a notificação *onDeltaSequenceEnd*, que informava os clientes do fim de um conjunto de operações sobre uma Wave. Esta notificação, usada pelo próprio *console client*, tinha especial interesse para uma versão inicial do módulo concebido, pois tornava o processamento de actualizações mais simples para os clientes da plataforma. Assim, teve-se que optar pelo uso directo da abstracção *ClientBackend* e implementar na interface portável as operações abstraídas pela *Agent* API.

Um ponto importante sobre o FedOne que é necessário referir é que, apesar de o servidor possuir o suporte para ter várias Wavelets por Wave, o *ClientBackend* e a *Agent* API apenas permitem que seja criada uma Wavelet por Wave. Considerou-se implementar nestes o suporte para ter múltiplas Wavelets por Wave, porém como uma Wave não tem qualquer conteúdo por si só e representa apenas um conjunto de Wavelets, concluiu-se que as vantagens não valiam o esforço. Como tal, todas as futuras referências neste relatório a uma Wave representam não só essa Wave como a Wavelet que a compõe.

4.1.2. A Interface Portável

Como foi visto no capítulo 3.1, os clientes da plataforma iriam comunicar com o módulo concebido através do protocolo Bayeux. Porém, era necessário escolher uma implementação do protocolo. Entre as implementações existentes em Java contava-se com a do Oracle WebLogic Server [57], IBM WebSphere [58] e CometD [59]. Destas optou-se pelo CometD, por ser a implementação de referência dos próprios autores do protocolo Bayeux.

```
public class WaveUA extends BayeuxService implements WaveletOperationListener{

    private final static long serialVersionUID = -2919167206889576860L;

    private ClientBackend agentConn;
    private final static ParticipantId agentId = new ParticipantId("WaveUA@ptin.pt");
    private final static String hostAddress = "10.112.128.214";
    private final static int hostPort = 9876;

    private Map<String,ClientBackend> waveClients;
    private Map<String,Client> bayeuxClients;

    public WaveUA(Bayeux bayeux) throws IOException {
        super(bayeux, "WaveUA");

        waveClients = new HashMap<String,ClientBackend>();
        bayeuxClients = new HashMap<String,Client>();

        agentConn = new ClientBackend(agentId.getAddress(), hostAddress, hostPort);
        agentConn.addWaveletOperationListener(this);

        subscribe("/service/command", "onRequest");
    }

    public void onRequest(Client remote, Map<String, Object> data) {
```

Listagem 1 – Implementação da Interface Portável

Na listagem 1 podemos ver a classe Java que representa o módulo concebido. A classe implementa a interface *WaveletOperationListener* para poder receber actualizações do servidor Wave e possui um membro privado do tipo *ClientBackend* para poder comunicar com o mesmo. No entanto esta não é a única ligação que o módulo mantém com o servidor. De modo a registar no Wave as operações pedidas pelos clientes sob os seus nomes, era necessário abrir uma ligação

entre este e cada cliente que se liga à interface, assim como efectuar as operações através dessas ligações. No entanto, como cada ligação iria despoletar na interface notificações de actualizações das diferentes Waves, o número de vezes que a interface receberia actualizações para cada Wave seria proporcional ao número de clientes que possui-se nessa Wave. Daí usar-se uma ligação adicional ao cliente Wave. Esta ligação representa a interface e, apesar de não ser usada para efectuar operações, é a única ligação usada para receber actualizações das Waves. Este registo é feito através do método *addWaveletOperationListener*.

Para usar o CometD e implementar a classe *WaveUA* como um servidor Bayeux, bastou que esta estendesse da classe *BayeuxService*. Para receber pedidos dos clientes a classe usa o paradigma *Publish/Subscribe* como especificado pelo protocolo Bayeux e subscreve do canal *"/service/command"*. Segundo o capítulo 2.2.3 da especificação do protocolo Bayeux [36], um canal com nomenclatura *"/service/***"* é um tipo especial de canal que apenas pode ser usado por servidores Bayeux para receber mensagens. Os clientes podem publicar mensagens neste tipo de canal, mas nunca podem subscrever e, consequentemente, receber mensagens do mesmo. Este tipo de canal é portanto ideal para implementar uma política de pedido/resposta e manter a privacidade dos pedidos efectuados por cada cliente. Tal política era necessária dada a natureza da interface: os clientes efectuavam pedidos de operações à interface e esta enviava-lhes a resposta.

Por fim, é de notar na listagem 1 que são guardadas duas estruturas de dados na classe *WaveUA*. A primeira, *waveClients*, guarda o nome e a ligação ao servidor Wave de cada utilizador que está actualmente ligado à interface portável. Uma alternativa a usar esta estrutura de dados seria abrir as ligações dos clientes apenas na duração necessária para efectuar cada pedido e fechá-la de seguida, repetindo este procedimento para cada pedido posterior. No entanto pensou-se que a aproximação escolhida teria melhor performance temporal, um requisito muito importante para colaboração em tempo real. A segunda estrutura de dados, *bayeuxClients*, guarda referências para os clientes Bayeux de forma a poder enviar-lhes actualizações das Waves em que são participantes quando recebidas na interface. A necessidade de guardar estas referências surge do uso do paradigma pedido/resposta e de só se comunicar com um cliente de cada vez. Uma alternativa seria usar o paradigma *Publish/Subscribe* e registar um canal para cada Wave, publicando actualizações nesses canais. No entanto esta aproximação poderia comprometer a privacidade das Waves pois qualquer cliente Bayeux poderia ligar-se à interface e receber actualizações de Waves a que não pertencia.

4.1.3. Operações da Interface

No contexto deste trabalho havia a necessidade de disponibilizar na interface apenas operações de interacção directa com as Waves. Toda a lógica de criação e obtenção das Waves

de um participante era feita pelo agendamento de sessões na plataforma, como descrito no capítulo 3.2.

```
Parâmetros comuns a todos os pedidos:
    - name :
        - O nome do cliente.
    - action :
        - A operação que se pretende efectuar. Operações disponíveis:
            - /login : - Operação inicial para se registar na interface.
            - /logout : - Operação final para sair da interface e libertar recursos.
            - /openwave : - Operação para obter conteúdo de uma wave
            - /addblip : - Operação para adicionar um blip no fim de uma Wave.

Parâmetros de entrada e saída das operações disponíveis:
    - /login
        - Entrada: { }
        - Saída:
            - ok : Login efectuado com sucesso.
            - fail : Se cliente já tiver feito login ou erro ao comunicar com servidor Wave
    - /logout
        - Entrada: { }
        - Saída:
            - ok : Logout efectuado com sucesso.
    - /openwave
        - Entrada:
            - waveId : id da wave a abrir.
        - Saída:
            - waveId wave : O id da wave seguido do seu conteúdo em JSON.
            - fail "excepção" : Se houver alguma excepção, seguida da mesma.
    - /addblip
        - Entrada:
            - waveId message : id da wave a adicionar o blip seguido do conteúdo do mesmo.
        - Saída:
            - waveId blipId : O id da wave seguido do id do blip criado.
            - fail "excepção" : Se houver alguma excepção, seguida da mesma.
```

Listagem 2 – Versão inicial da interface

A listagem 2 representa as operações a disponibilizar na interface após um levantamento de requisitos inicial. Como se estava a usar o código do *console client* e da *Agent* API do FedOne como exemplo, as operações disponibilizadas na interface estavam limitadas pelas operações implementadas nos mesmos. Como estes estavam bastante incompletos face ao serviço Google Wave, seguiu-se a especificação *Conversation Model* [60] para implementar as restantes funcionalidades no projecto do servidor FedOne. Um segundo levantamento de requisitos destas funcionalidades originou as novas operações na interface, como descrito na listagem 3. Para simplificar a listagem foi omitido de todas as operações o possível retorno de excepção em caso de erro.

```
- /deleteBlip
    - Entrada:
        - waveId blipId: id da wave seguido do id do blip a remover.
    - Saída:
        - waveId blipId : O id da wave seguido do id do blip removido.
- /addBlipAfter
    - Entrada:
        - waveId blipId message: id da wave seguido do id do blip anterior e da mensagem do novo blip.
    - Saída:
        - waveId blipId : O id da wave seguido do id do blip criado.
- /addBlipBefore
    - Entrada:
        - waveId blipId message: id da wave seguido do id do blip seguinte e da mensagem do novo blip.
    - Saída:
        - waveId blipId : O id da wave seguido do id do blip criado.
- /addReplyBlip
    - Entrada:
        - waveId blipId message: id da wave seguido do id do blip ao qual se está a responder e da mensagem do novo blip.
    - Saída:
        - waveId blipId : O id da wave seguido do id do blip criado.
- /addInlineReplyBlip
    - Entrada:
        - waveId blipId index message: id da wave seguido do id do blip ao qual se está a responder,
        do indice dentro desse blip onde se quer inserir a resposta e da mensagem do novo blip.
    - Saída:
        - waveId blipId : O id da wave seguido do id do blip criado.
- /insertTextInBlip
    - Entrada:
        - waveId blipId index message: id da wave seguido do id do blip que se está a editar,
        do indice dentro desse blip onde se quer inserir o texto e do texto a inserir.
    - Saída:
        - waveId blipId : O id da wave seguido do id do blip criado.
- /deleteTextFromBlip
    - Entrada:
        - waveId blipId index message: id da wave seguido do id do blip que se está a editar,
        do indice dentro desse blip onde se quer remover texto e do texto a remover.
    - Saída:
        - waveId blipId : O id da wave seguido do id do blip criado.
```

Listagem 3 – Novas Operações da Interface

4.1.4. Servidor Web

Terminado o servidor Bayeux com a interface concebida, era necessário disponibilizá-lo aos clientes por HTTP. Segundo a documentação do projecto CometD [61], tal deveria ser feito através da definição de uma Servlet [62]. Para tal usou-se uma classe auxiliar que estendia de *GenericServlet* e iniciava o serviço Bayeux. Esta classe, juntamente com a classe *WaveUA* e outras classes auxiliares, foram agrupadas num *Web Application Archive* (WAR) devidamente configurado. Por fim, este WAR deveria ser *deployed* num *Servlet Container* [62] de modo a ser disponibilizado na Web. Para tal usou-se o Jetty [63], por ser dos mesmos criadores do Bayeux e CometD e por ser referido na documentação do projecto CometD que este estava optimizado para ser usado em conjunto com o Jetty e era mais escalável nesta situação.

4.2. Integração do Wave na Plataforma

No capítulo 3.2 foi apresentada a solução concebida para esta contribuição. Tal solução consistia em adicionar um servidor Wave à plataforma como novo recurso e desenvolver todo o

suporte necessário do lado da mesma para a sua utilização. Este capítulo irá apresentar como foi implementado esse suporte.

Como vimos no capítulo 4.1.1, o servidor Wave escolhido foi o FedOne. Para fazer a ligação entre este e a plataforma pensou-se usar a mesma aproximação que nesse capítulo. No entanto, como neste caso as notificações geradas pela *Agent* API eram suficientes, optou-se por usar esta. Mais especificamente, havia duas notificações do servidor Wave que interessava receber nesta contribuição: notificação de participante adicionado e de participante removido.

A listagem 4 representa o componente *WaveConnector*. Como foi visto no capítulo 3.2, este é o componente que faz a ligação com o servidor Wave. Para tal, a classe estende da classe *AbstractAgent*. Esta é a forma estandardizada de usar a *Agent* API e definir um novo agente. A classe implementa ainda a interface *WaveConnectorLocal*, usada internamente na plataforma para efectuar operações nos recursos de Wave. As operações desta interface incluem a criação e obtenção de uma Wave, adição e remoção de participantes de uma Wave e o envio de novos Blips para uma Wave. De notar que para iniciar o funcionamento de um agente do Wave é necessário invocar o método *run()* da classe *AbstractAgent*. Porém este método é bloqueante, pois inicia um ciclo infinito de recepção de eventos. Como tal, e de forma a não bloquear o funcionamento de toda a plataforma, a invocação deste método foi feita num novo *thread*.

```
@Service
@Management(WaveConnectorLocal.class)
@Local(WaveConnectorLocal.class)
public class WaveConnectorBean extends AbstractAgent implements WaveConnectorLocal {

    @EJB
    private ResourceManagerLocal resourceManager ;

    public void start() throws Exception {
        log.info("[start] WaveConnector starting...") ;
        new Thread(new Runnable() {
            public void run() {
                startAgent() ;
            }
        }).start() ;
        log.info("[start] WaveConnector started...") ;
    }

    private void startAgent() {
        super.run() ;
    }
```

Listagem 4 – Implementação do componente *WaveConnector*

Para implementar este e os outros componentes da solução concebida usou-se a tecnologia das bibliotecas J2EE [64] e do JBoss AS 5.1 [65]. Estas tecnologias são também usadas nos outros módulos da plataforma, estando assim os componentes implementados para esta contribuição em conformidade. Dessas tecnologias, escolheu-se o *Singleton* para implementar o *WaveConnector*. Um *Singleton* é uma classe Java que em qualquer altura terá sempre uma e

apenas uma instância, podendo esta ser acedida globalmente. Apesar de ainda não existir na especificação EJB 3.0 [66] este tipo de *Bean*, o seu comportamento pode ser simulado pelo uso de um *Service Bean* do JBoss [67]. Tal é especificado com a anotação *@Service*, por cima da declaração da classe. Adicionalmente, a anotação *@Local* especifica a interface que é usada por outros *Beans* da plataforma para invocar localmente o *WaveConnector*. Ainda segundo [67], o método *start()* é invocado quando o *Bean* inicia os seus serviços. Como tal, escolheu-se usar este método para iniciar o agente num novo *thread*.

Quando o *WaveConnector* recebe uma notificação de participante adicionado ou removido, é responsabilidade deste enviar o evento correspondente para o *ResourceController* certo. Para tal, invoca-se o método *getResourceSessionByRoomLocation* do *Bean ResourceManager*, passando-lhe o *id* da Wave. Este método acede à base de dados, pesquisa pelo atributo *RoomLocation* da entidade *ResourceSession* e devolve um *ResourceController* instanciado com o *ResourceSession* obtido.

A listagem 5 representa o outro componente desenvolvido nesta contribuição, o *WaveController*. Este componente implementa a interface *TextResourceController*, que por sua vez estende a interface *ResourceController*. Esta interface representa um controlador de recursos genérico e a interface *TextResourceController* adiciona-lhe alguns métodos específicos ao tipo de recurso de texto. O *WaveController* estende ainda de *ResourceControllerAbstract*, uma classe abstracta que implementa a interface *ResourceController* e implementa alguns dos seus métodos que são comuns a todos os tipos de controladores de recursos.

```
@Stateless
@Local(TextResourceController.class)
public class WaveControllerBean extends ResourceControllerAbstract implements TextResourceController {

    @EJB
    private WaveConnectorLocal connector ;
```

Listagem 5 – Implementação do componente *WaveController*

O *Bean WaveController* foi implementado como um *Stateless* [66]. Um *Stateless Bean* é um tipo de *Bean* que não guarda qualquer estado. Instâncias de *Stateless Beans* são criadas por invocação pelo EJB *Container*, neste caso o servidor aplicacional JBoss, e geralmente são lhes passadas em cada método as entidades onde vão operar. No entanto, como os *ResourceControllers* estendem de uma classe abstracta, esta classe pode guardar a entidade onde vão operar, o *ResourceSession*. Assim, no método *getResourceSessionByRoomLocation* do *Bean ResourceManager* o *ResourceController* instanciado é logo afectado com o *ResourceSession* obtido da base de dados e só depois devolvido.

A interface *ResourceController* continha vários métodos colaborativos genéricos. No entanto nem todos faziam sentido ser implementados no contexto do Wave. Os métodos que foram implementados no *WaveController* são:

- *setResources*: método invocado quando é feito o *setup* da sessão. Neste método o *WaveController* comunica com o *WaveConnector* para criar uma nova Wave, cria uma nova entidade *ResourceSession*, afecta os seus vários atributos incluindo o *RoomLocation* e por fim cria um novo *ResourceElement* do tipo *Text*, correspondente à Wave criada. A classe *Text* referida é uma nova entidade criada nesta contribuição. Esta entidade estende da entidade *ResourceElement* e representa um recurso de texto que foi adicionado à plataforma.
- *newResourceClient*: método invocado quando é adicionado novo participante à sessão. Como resposta o *WaveController* cria um novo *ResourceClient*, afecta os seus atributos e se a sessão estiver aberta adiciona o participante à Wave, através do *WaveConnector*.
- *Disconnect*: método invocado quando um participante é desconectado da sessão. Neste método o *WaveController* comunica com o *WaveConnector* e pede-lhe para remover o participante da Wave.
- *processEvent*: método invocado pelo *WaveConnector* quando este recebe um evento do servidor Wave. Neste método o *WaveController* comunica com o *Bean* que opera sobre a entidade da sessão, o *SessionManager*, e reenvia-lhe o evento sem o alterar. A referência para este *Bean* é guardada na classe *ResourceControllerAbstract*.
- *updateStatus*: método invocado quando o estado da sessão é alterado. No contexto do Wave, existem três estados que podem ser afectados: *Open*, *Pause* e *Close*. Quando a sessão é aberta (*Open*), o *WaveController* pede ao *WaveConnector* para adicionar os participantes da sessão à Wave respectiva. Assim garante-se que a Wave não pode ser alterada antes de a sessão ser aberta. No caso de pausar (*Pause*) ou fechar (*Close*) a sessão, os participantes são removidos da Wave. Garante-se assim que nestes dois estados a Wave não pode ser alterada. No entanto, no caso de a sessão ter sido pausada ela pode novamente ser aberta, enquanto que quando fechada os participantes não podem entrar novamente na Wave. Pode-se argumentar que assim os participantes deixam de ter acesso ao documento editado quando fechada a sessão. Porém este problema é resolvido na solução concebida no capítulo 3.3, com a adição de funcionalidades de *Recording* e *Play* à plataforma.

4.2.1 User In/Out no Wave

Segundo a especificação da plataforma PUC, quando um utilizador se liga ou desliga de um servidor de recursos este deve notificar a plataforma do mesmo. Desta forma a plataforma consegue centralizar o estado de cada utilizador nos vários servidores de recursos e informar as aplicações cliente do mesmo.

No caso do servidor Wave usado, o FedOne [4] (ver cap. 4.1.1), este evento não existia. Apesar de o servidor ser capaz de registar quando um utilizador se ligava ou desligava, tal informação não era guardada.

Para implementar o envio destes eventos havia duas hipóteses a considerar. A primeira era implementar o envio directamente no servidor. Esta era a hipótese mais completa pois todas as aplicações cliente do servidor Wave poderiam usufruir do evento. A outra hipótese seria implementar o envio dos eventos na interface portável que tinha sido criada no capítulo 4.1.2. Esta solução tinha a vantagem de, para além de ser mais simples de implementar, permitir manter o servidor Wave sem qualquer alteração, simplificando a sua actualização. Porém, com esta solução apenas os clientes que usassem a interface portável poderiam usufrui dos novos eventos. Depois de uma reflexão sobre as vantagens e desvantagens das duas hipóteses, optou-se por implementar a primeira.

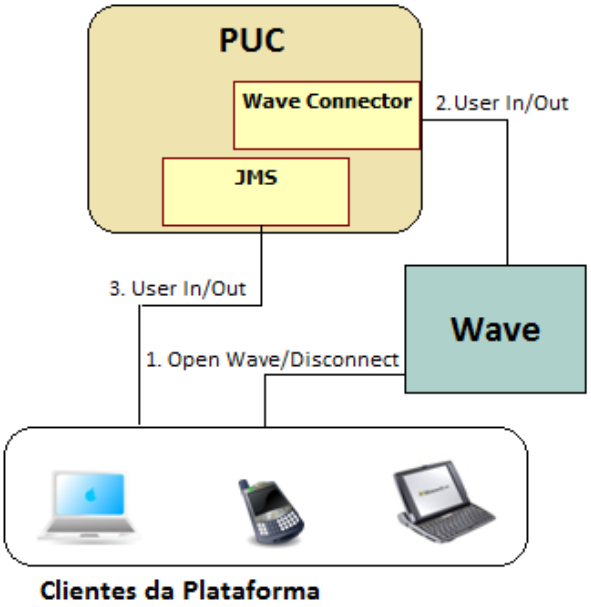


Figura 12 - Arquitectura e sequência de envio de eventos User In/Out

A figura 12 mostra a arquitectura da solução desenvolvida, assim como uma sequência de envio dos eventos. Quando um utilizador abre uma Wave, através de qualquer um dos seus clientes, é enviada uma notificação a todos os utilizadores dessa Wave que este acabou de entrar na mesma (User In). Na plataforma o componente WaveConnector recebe este evento e envia-o para os outros componentes da plataforma para ser processado. Por fim é usado o sistema de eventos da plataforma para enviar o evento aos seus clientes (este sistema, cuja arquitectura foi discutida em 3.3, será explicado em mais detalhe no cap. 4.3.2).

Juntamente com os dados da ligação que já eram guardados no servidor Wave para cada cliente, é agora também guardada a Wave que o cliente tem aberta num dado momento. Assim, quando o utilizador muda de Wave ou se desconecta do servidor, é gerado o evento de User Out para a Wave que o utilizador estava a editar, ou seja, os outros clientes são informados que o utilizador se desligou dessa Wave. O processamento desse evento segue o fluxo descrito para o evento de User In.

4.3. Funcionalidades de *Recoding* e *Play*

Este capítulo irá apresentar como foi implementada a solução concebida no capítulo 3.3. Resumindo esse capítulo, a solução dividia-se em duas fases. Primeiro iria ser adicionado um novo módulo ao modelo de dados da plataforma, de forma a suportar o *Recording* dos recursos. Ainda no contexto desta primeira fase seria também desenvolvido um sistema para efectuar a entrega assíncrona de eventos dos recursos às aplicações da plataforma. Na segunda fase seriam desenvolvidas as funcionalidades de *Play* para o recurso Wave, com o *import* e *export* de documentos Wave para outros formatos.

4.3.1. *Recording*

A solução concebida para este problema consistia em adicionar um novo componente ao modelo de dados da plataforma. A listagem 6 representa esse componente, a classe *SessionDocument*. A classe foi implementada como uma entidade, como indica a anotação *@Entity*. Segundo a especificação EJB 3.0 [66], uma *Entity* ou entidade é um tipo de *Bean* geralmente usado para implementar directamente os componentes de um modelo de dados. A entidade caracteriza-se também por ser *long-lived* e pelo seu estado ser persistido pelo EJB *Container*, através do uso de uma base de dados.

A classe possui também a anotação *@Table* para definir o nome que a tabela criada na base de dados deve ter. Como atributos, a classe *SessionDocument* possui um *id*, usado internamente pela base de dados, e uma referência para a entidade *Session*, a entidade que representa a sessão. Este atributo possui uma anotação *@OneToOne*, pois como foi visto no capítulo 3.3, cada sessão tem um e um só *SessionDocument* e cada *SessionDocument* pertence a uma única sessão.

Como explicado no capítulo 3.3, a entidade possui mais três atributos. O primeiro é um mapa de *Strings* e representa as *utterances* da sessão, com o *id* de cada uma como chave e o documento da *utterance* como valor. Para persistir este atributo na base de dados usa-se a anotação *@CollectionOfElements*. Como o tamanho dos documentos XML das *utterances* podia ser bastante elevado, foi necessário alterar o tamanho da coluna deste atributo na base de dados. Tal foi feito através da anotação *@Column*.

```java
@Entity
@Table(name = "puc_session_document")
public class SessionDocument implements Serializable {

    @Id
    @GeneratedValue
    private int id ;

    @OneToOne(fetch = FetchType.EAGER)
    private Session session ;

    @CollectionOfElements(fetch = FetchType.EAGER)
    @Column(length=Integer.MAX_VALUE)
    private Map<String, String> utterances ;

    @Enumerated(EnumType.STRING)
    private UtteranceGuideType utteranceGuide ;

    private boolean textResourceSynch ;

    private String previousUtteranceId ;
```

Listagem 6 – Implementação do componente *SessionDocument*

Apesar de ser necessário guardar os documentos XML das *utterances* como Strings de forma a poderem ser persistidos, para operar mais facilmente sobre eles usa-se a biblioteca Java para o DOM [48]. O que isto significa é que quando se pretende editar uma *utterance* esta é primeiro convertida de String para *Document* do DOM, é então editada e por fim é convertida novamente para String e persistida.

O segundo atributo, implementado como uma enumeração, é o *utteranceGuide*. Como foi referido, este atributo define o critério de separação das *utterances*. O terceiro atributo apresentado no capítulo 3.3 e constante na listagem 6 é o *textResourceSynch*, implementado como um booleano e usado para definir se as *utterances* da sessão devem ser sincronizados com os seus recursos de texto. Como o booleano é um tipo primitivo em Java, o atributo é automaticamente persistido sem ser necessária qualquer anotação.

Por fim, a listagem 6 mostra um último atributo, *previousUtteranceId*. Esta *String*, que guarda o *id* da última *utterance* criada, é um atributo auxiliar necessário de modo a efectuar correctamente a progressão de *utterances*. O atributo tinha que existir na entidade *SessionDocument*, caso contrário não seria persistido e seria perdido entre invocações do *Stateless* que contém a lógica de operações sobre esta entidade, como será visto a seguir. Chegou-se a considerar se este atributo seria realmente necessário e se não se poderia usar o *Manifest* para descobrir o último *utterance* criado. No entanto, como foi referido no capítulo 3.3, a relação entre as *utterances* e consequentemente a estrutura do *Manifest* não tinha que ser obrigatoriamente sequencial, tendo inclusivamente sido visionado e implementado o suporte para *utterances* criadas como resposta a outras já existentes. Como tal, tornava-se impossível saber a partir do *Manifest* qual a última *utterance* criada.

O objectivo de uma entidade é definir uma tabela na base de dados e suas colunas. Como tal, uma entidade apenas deve conter métodos para aceder e alterar os seus atributos. Para definir a lógica de operação sobre essa tabela deve-se usar outra categoria de *Beans*, conhecida como *Session Beans* [66]. Desta categoria, tem especial interesse o *Stateless Session Bean*, por não guardar qualquer estado e como tal ser mais leve para a performance da plataforma.

```
@Stateless
@Local(UtteranceManagerLocal.class)
@Remote(UtteranceManagerRemote.class)
public class UtteranceManagerBean implements UtteranceManagerLocal, UtteranceManagerRemote {
    private static Logger log = Logger.getLogger(UtteranceManagerBean.class) ;

    @EJB
    private SessionDocumentService sessionDocumentService ;

    @EJB
    private ResourceManagerLocal resourceManager ;
```

Listagem 7 – Implementação do componente *UtteranceManager*

A listagem 7 representa o *Stateless Bean* desenvolvido para operar sobre a entidade *SessionDocument*. Para além da anotação *@Stateless*, a classe possui mais duas anotações, *@Local* e *@Remote*. Estas anotações são usadas para definir, respectivamente, a interface que pode ser usada internamente por outros *Beans* da plataforma e a interface que pode ser invocada pelas aplicações da plataforma. Pode-se ainda ver que a classe *UtteranceManagerBean* comunica com mais dois componentes da plataforma, o *ResourceManager* e o *SessionDocumentService*. O *ResourceManager* é usado para obter os *ResourceControllers* quando é necessário operar sobre os recursos. Exemplo disto é a invocação do inicio de gravação das *streams* da sessão aquando do inicio de uma nova *utterance*. Outro exemplo é a sincronização com os recursos de texto no fim de uma *utterance*. O *SessionDocumentService* é um *Stateless* criado para persistir o *SessionDocument* e obtê-lo da base de dados. Como tal, é apenas usado pela classe *UtteranceManagerBean*.

O *UtteranceManager* possui os métodos necessários para implementar os casos de uso especificados no capítulo 3.3. Estes incluem: *setUtteranceGuide*, que inicia uma nova sequência de *utterances* guiadas pelo guia passado como argumento; e os métodos *startManualUtterance* e *stopManualUtterance*, para respectivamente iniciar e parar *utterances* manualmente. Estes três métodos, juntamente com o método *setTextResourceSynch*, compõe a interface remota do *Bean*, ou seja, são os métodos que podem ser invocados pelos clientes.

4.3.2. Envio de Eventos

Como foi referido no capítulo 3.3, para se poder fazer a progressão de *utterances* era necessário um mecanismo que permitisse o envio de eventos dos recursos para as aplicações do PUC. Um novo componente deste módulo iria depois usar o mecanismo desenvolvido para

receber os eventos referidos. No capítulo 3.3 foi decidido que seria usado um sistema MOM para efectuar este envio. A principal razão desta escolha devia-se à natureza assíncrona destes sistemas, que permitia grande desacoplamento entre a plataforma e as aplicações clientes.

Actualmente existem diversas implementações de sistemas de MOM. Numa tentativa de unificar todas as diferentes implementações sob uma interface comum, foi criada a Java Message Service (JMS) [68]. Para este trabalho tornava-se especialmente interessante usar um sistema que implementasse esta API pois toda a plataforma tinha sido criada com uso da linguagem Java e das tecnologias da *Framework* J2EE. Entre as diferentes implementações de JMS, optou-se pelo Apache ActiveMQ [69]. O Apache ActiveMQ possuía diversas funcionalidades de integração e suportava diversos protocolos. Algumas destas funcionalidades tinham interesse imediato para este trabalho, como a integração com o servidor aplicacional JBoss. Outras funcionalidades tinham um interesse potencial futuro, no desenvolvimento da plataforma posterior a este trabalho.

No entanto, a principal razão de escolha do Apache ActiveMQ foi por permitir a criação de tópicos e filas dinamicamente. Segundo a especificação JMS [68], tópicos e filas são objectos que devem ser criados administrativamente. Porém, para este trabalho isso implicava que apenas poderia haver um destino (tópico ou fila) na plataforma, e este seria partilhado por todas as sessões colaborativas e por todos os participantes. De forma a poder distribuir a carga por diferentes destinos e de forma a também manter alguma privacidade, interessava ter um destino para cada sessão da plataforma. Contudo, como as sessões são criadas dinamicamente pelos clientes da plataforma, para haver um destino por sessão estes teriam também que poder ser criados dinamicamente. Esta aproximação tinha como vantagens permitir distribuir a carga das mensagens trocadas entre diferentes destinos e apenas permitir que os participantes de uma sessão pudessem ver as mensagens trocadas através do destino da mesma.

```
@Service()
@Management(BrokerConnectorLocal.class)
@Local(BrokerConnectorLocal.class)
public class BrokerConnectorBean implements BrokerConnectorLocal {

    private MBeanServerConnection connection ;
    private ObjectName brokerViewMBeanName ;

    private Connection conn ;
    private Session session ;
    private MessageProducer publisher ;

    private Map<String,UtteranceMessageListener> messageListeners;
```

Listagem 8 – Implementação do componente que comunica com o *broker* de JMS

Era também necessário escolher o tipo de destino a usar, ou seja, se seriam usados tópicos ou filas. Dos dois escolheu-se usar tópicos pois estes correspondiam melhor à natureza das sessões colaborativas: os clientes subscreviam do tópico da sessão em que eram participantes e,

quando surgissem eventos nos recursos da mesma, a sessão publicava-os no seu tópico e todos os clientes que tivessem feito a subscrição recebiam os eventos.

Para fazer a ligação entre a plataforma e o *broker* do ActiveMQ foi desenvolvido um novo componente. A listagem 8 representa este componente, denominado de *BrokerConnector*. O objectivo do componente é de abstrair todos os outros módulos da plataforma dos detalhes de comunicar com o *broker* de JMS, assim como manter um único ponto de comunicação entre os dois. Para tal, o *BrokerConnector* foi implementado como um *Singleton*, através da interface *@Service*, muito à semelhança de como foi implementado o *WaveConnector* no capítulo 4.2. Como nesse componente, a anotação *@Local* indica a interface usada pelos outros *Beans* da plataforma para invocar os seus métodos. Os métodos desta interface são:

- *createTopic*: método que cria um tópico novo dinamicamente. O novo tópico é identificado pela String passada como argumento. Este método é invocado pelo *SessionManager* quando é feito o setup de uma sessão, passando-lhe o identificador da sessão.
- *closeTopic*: método que remove dinamicamente um tópico do *broker*, dado o seu identificador. Este método é invocado pelo *SessionManager*, quando uma sessão é fechada.
- *publishEvent*: método que recebe um evento e publica-o no tópico da sessão a que pertence. O evento recebido é uma classe Java criada pelo *SessionManager* a partir do evento disparado dos recursos, ao qual adiciona informações como o *id* da sessão.

Como atributos, o *BrokerConnectorBean* possui os necessários para comunicar com o *broker* de JMS. Estes são uma *Connection*, uma *Session* e um *MessageProducer*. São ainda necessários mais dois atributos do pacote *javax.management*, *MBeanServerConnection* e *ObjectName*, para remover um tópico do *broker*. Esta necessidade surge da inexistência de um método na API do JMS para efectuar essa operação e, como tal, ser preciso aceder directamente ao *Bean* do *broker* de ActiveMQ. Por fim, o *BrokerConnectorBean* possui um mapa de *UtteranceMessageListeners*. Esta é a classe Java que subscreve dos tópicos das sessões e, ao receber um evento, analisa se o *SessionDocument* correspondente está à espera do mesmo. Em caso positivo, e dependendo do tipo de evento, invoca o método certo do lado *Stateless UtteranceManager*. Este mapa é necessário porque é preciso guardar uma referência para cada *UtteranceMessageListener* de modo a que estes não sejam removidos pela *garbage collection* do Java. A razão de ser o *BrokerConnector* a guardar este mapa é que como é um *Singleton*, é o único *Bean* capaz de assumir esta responsabilidade. O *UtteranceManager*, como é um *Stateless*, nunca poderia guardar um estado. Por outro lado, o *SessionDocument*, na qualidade de entidade, só poderia guardar esta classe se ela pudesse ser persistida, o que não acontece.

A listagem 9 representa a classe *UtteranceMessageListener*. Para cada *UtteranceMessageListener* é criada uma JMS *Session* e a partir da mesma um JMS *MessageConsumer*. O uso de uma sessão de JMS para cada *UtteranceMessageListener* garante a

independência na recepção de mensagens de cada um. O uso de *MessageConsumers* diferentes é necessário uma vez que estes só podem estar associados a um destino de cada vez. O *UtteranceMessageListener* possui ainda uma referência para a interface local do *UtteranceManager*. Esta é usada para invocar as operações sobre o *SessionDocument* da sessão quando são despoletados os eventos certos, assim como para obter o próprio *SessionDocument*.

```
public class UtteranceMessageListener implements MessageListener {

    @EJB
    private SessionManagerLocal sessionManager ;

    @EJB
    private UtteranceManagerLocal utteranceManager ;

    private MessageConsumer messageConsumer;

    private javax.jms.Session jmsSession;
```

Listagem 9 – Implementação do componente *UtteranceMessageListener*

A necessidade de haver uma colecção de *UtteranceMessageListeners* no *BrokerConnectorBean* é consequência do uso de múltiplos tópicos criados dinamicamente. Se só houvesse um tópico partilhado por todas as sessões era apenas necessário haver um *MessageConsumer* de JMS. Por outro lado, mesmo havendo múltiplos tópicos, se estes não fossem criados dinamicamente poderia ser usado um tipo de *Bean* chamado *Message-Driven Bean* [66]. O uso deste *Bean* simplificaria o processo de recepção de mensagens e tornaria a existência da colecção desnecessária.

4.3.3. Optimizações do Sistema de Eventos

Depois das avaliações experimentais de desempenho realizadas sobre a plataforma (explicadas em detalhe nos caps. 5.2.2, 5.2.3 e 5.2.4), foi feito um levantamento de optimizações dos sistemas desenvolvidos que deveriam ser realizadas de forma a melhorar o desempenho das mesmas. Entre estas listavam-se optimizações do sistema de envio de eventos da plataforma. Este capítulo descreve as optimizações que foram implementadas e como foram realizadas.

A primeira optimização que foi implementada incidiu sobre a quantidade de tópicos JMS [68] que eram utilizados. Originalmente, quando era criada uma sessão colaborativa era criado dinamicamente um tópico JMS que seria usado apenas para os eventos desta. Com esta optimização, em vez de haver múltiplos tópicos criados dinamicamente, passa a existir apenas um criado estaticamente quando a plataforma é iniciada. Este tópico é partilhado por todas as sessões da plataforma. Com esta optimização erspera-se tornar o processo de criação de sessões mais rápido.

Para além dos resultados que esta optimização traz só por si, ela também abre o caminho para outras optimizações ainda mais relevantes. Uma destas é a mudança de implementação de JMS usada. Como já não há necessidade de criar tópicos dinamicamente, é possível abandonar o sistema ActiveMQ [69] e passar a utilizar o sistema de JMS que vem incorporado com o servidor aplicacional JBoss [71], o JBoss Messaging [81]. Com esta optimização consegue-se poupar na memória necessária para manter a plataforma em funcionamento.

A última optimização que passou a ser possível realizar incidiu sobre o componente *UtteranceMessageListener* (cap. 4.3.2). Como os tópicos da plataforma eram criados dinamicamente, as instâncias deste módulo tinham que ser criadas manualmente e guardadas no módulo *BrokerConnector* aquando da criação dos tópicos. Com a passagem a um único tópico criado estaticamente, passou a ser possível implementar este módulo como um *Message-Driven Bean* [66]. Assim, todo o ciclo de vida das suas instâncias é gerido pelo servidor JBoss, facilitando a gestão do mesmo.

```
@MessageDriven(activationConfig = {
        @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Topic"),
        @ActivationConfigProperty(propertyName = "destination", propertyValue = "/topic/PUCTopic") })
public class UtteranceMessageListener implements MessageListener {

    private static Logger log = Logger.getLogger(UtteranceMessageListener.class) ;

    @EJB
    private SessionManagerLocal sessionManager ;

    @EJB
    private UtteranceManagerLocal utteranceManager ;
```

Listagem 10 - Componente *UtteranceMessageListener* optimizado

A listagem 10 representa o componente *UtteranceMessageListener* depois da optimização, juntamente com o cabeçalho que o define como um *Message-Driven Bean*. Comparando-o com a sua implementação original, nota-se que este já não precisa de guardar nem gerir um *MessageConsumer* e uma *Session* de JMS, pois tal é feito pelo servidor aplicacional.

4.3.4. Eventos da Plataforma em Dispositivos Heterogéneos

Para garantir fiabilidade e assincronismo, o sistema concebido para entrega de eventos da plataforma aos seus clientes baseou-se no método de comunicação publish/subscribe, nomeadamente no sistema JMS [68]. No entanto, os clientes móveis da plataforma revelaram-se incapazes de receber e processar as mensagens deste sistema. Como tal, e no âmbito do suporte à heterogeneidade na plataforma, revelou-se necessário desenvolver um módulo para conversão e entrega destes eventos aos referidos clientes.

A figura 13 mostra a arquitectura da solução concebida. Como se pode observar, para entregar os eventos aos clientes móveis aproveitou-se o conhecimento já adquirido no capítulo 2.4 sobre comunicações bidireccionais em ambientes Web. A solução concebida consistiu,

portanto, em desenvolver uma Servlet Bayeux [36] que fizesse de intermediária entre os clientes e a plataforma. Como implementação do protocolo Bayeux usou-se o CometD [59].

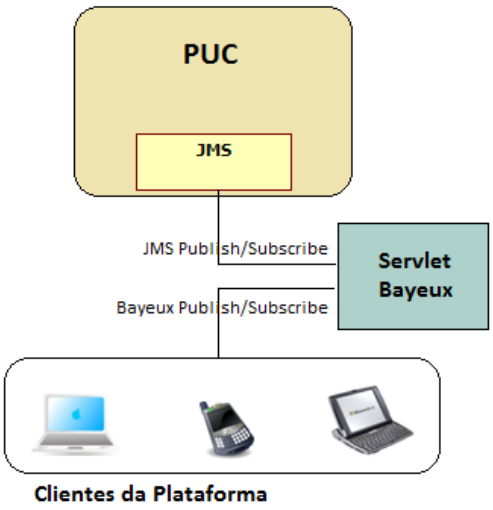


Figura 13 – Eventos em dispositivos heterogéneos

Explica-se de seguida o funcionamento da solução: A Servlet regista-se no sistema de JMS da plataforma e subscreve do tópico que é partilhado por todas as sessões colaborativas. Ao receber eventos no formato JMS, a Servlet converte-os em mensagens Bayeux e publica-os em canais Bayeux, identificados pelo id da sua sessão colaborativa. Tal é feito porque no CometD não é necessário criar canais *a priori*, ou seja, pode-se subscrever de ou enviar mensagens para canais sem ter que os criar explicitamente. Assim simplifica-se também os clientes pois só recebem os eventos das sessões em que estão interessados.

A conversão de mensagem JMS para Bayeux foi optimizada de forma a só converter as informações e dados usados na descrição do evento da plataforma. No entanto, a conversão é quase directa pois os modelos de dados usados por ambos são em muito semelhantes. Assim sendo, a solução concebida poderia facilmente ser adaptada de forma a converter todos os campos e o corpo completo das mensagens JMS no correspondente da mensagem Bayeux. Desta forma teríamos um sistema genérico de publicação de mensagens e eventos capaz de ser usado por dispositivos móveis e clientes Web.

4.3.5. Play

Implementadas as soluções de *Recording* concebidas no capítulo 3.3, faltava implementar as soluções para *Play* do recurso Wave. Como foi visto no capítulo 3.3, para importar e exportar documentos do Wave iria ser usado o formato HTML como formato interno da plataforma. Para

importar/exportar de/para outros formatos, estes teriam que ser primeiro convertidos para/de HTML.

A primeira tarefa a realizar era portanto implementar um conversor de HTML para o formato dos documentos do Wave. O formato do Wave e as suas possíveis anotações encontram-se descritos em [60], assim como os elementos HTML existentes são descritos em [70]. Estes documentos foram usados para traduzir os elementos HTML para elementos Wave com o mesmo significado, e vice-versa.

Para converter de HTML para Wave era necessário processar o documento HTML do inicio ao fim, convertendo anotação por anotação. Como o HTML [46] é uma aplicação da linguagem SGML [47] e o XML [45] é um *subset* da mesma, geralmente usa-se *parsers* de XML para processar HTML. Actualmente existem duas formas predilectas de processar XML, através do Document Object Model (DOM) [48] e através do Simple API for XML (SAX) [49]. O DOM permite mapear um documento XML numa estrutura de dados em árvore e aceder ao seu conteúdo. No entanto pode ter custos bastante elevados, principalmente em termos de memória. Por outro lado, o SAX permite processar um documento XML de uma forma serializada, como se este fosse uma *stream*, disponibilizando uma API *Event-Driven*. Este método de processar documentos, para além de ser mais leve e usar menos recursos, permite mais facilmente processar um documento XML do princípio ao fim e efectuar operações quando são detectados elementos do documento. Como isto descrevia exactamente o que se pretendia fazer neste trabalho, decidiu-se construir um SAX *Parser* de forma a converter de HTML para Wave.

Das diferentes implementações *open source* existentes da API SAX, optou-se por usar a biblioteca Apache Xerces [50], por ser da fundação Apache e por vir incluída nas bibliotecas do JBoss 5.1. Desta forma reduzia-se o número de bibliotecas que era necessário incluir na distribuição da plataforma. Para construir um *Parser* SAX criou-se uma classe a estender da classe *DefaultHandler*. Esta classe define métodos *callback* que são invocados quando são detectados certos elementos no documento XML. Estes métodos são:

- *startElement*, invocado quando é detectada a abertura de uma *tag*. A *tag* é convertida para a anotação Wave correspondente, segundo [60].
- *endElement*, invocado quando é detectado o fecho de uma *tag*. A anotação Wave correspondente é fechada.
- *characters*, invocado quando são detectados caracteres no documento. Os caracteres são escritos no documento Wave.

Na execução dos métodos da classe *callback* vão se juntando operações Wave a um *DocOpBuilder* para construir o Blip com o HTML convertido. Esta classe *DocOpBuilder* pertence ao modelo de dados do Wave e é incluída no projecto do servidor FedOne. No fim, invoca-se o método *build* da mesma e é obtido o documento Wave pronto para ser enviado para

o servidor. Assim, um documento HTML importado dá origem a um documento Wave, ou seja, um Blip singular.

Para exportar um documento Wave para um documento HTML, usa-se uma metodologia semelhante à usada no processo de importação. Através do uso da classe *DocOpCursor*, também do modelo de dados do Wave, define-se uma classe *callback* para documentos Wave. Dessa classe, os métodos que interessam para este trabalho são:

- *elementStart*: invocado quando é detectada a abertura de uma *tag* no documento Wave. A única *tag* que interesse para a exportação de documentos é a *tag* <*line*> [60];
- *characters*: invocado quando são detectados caracteres no documento Wave. Estes são adicionados ao documento HTML;
- *annotationBoundary:* invocado quando é detectado o inicio, mudança ou fim de uma única ou um conjunto de anotações Wave. Dependendo do tipo de anotação, é aberta ou fechada uma *tag* no documento HTML.

A classe *callback* é aplicada ao documento Wave que se pretende exportar e, como resultado, é retornada uma String com o documento HTML convertido.

Os métodos para conversão de HTML para Wave e de Wave para HTML foram adicionados à classe criada no capítulo 4.1.3 para incluir as funcionalidades adicionadas ao servidor FedOne. Para os clientes poderem usar estas funcionalidades, foram adicionados dois novos métodos ao *Stateless Bean* que opera sobre a sessão, o *SessionManagerBean*, assim como na sua interface remota. Estes métodos são exactamente o *importText* e o *exportText*. O primeiro recebe como argumento um *Media*. Esta é uma entidade da plataforma usada para representar um media, isto é, um ficheiro de áudio, um ficheiro de texto, etc. A partir do *Media*, o *SessionManager* obtém o url donde pode obter o documento e obtém o formato do documento (HTML, DOCX, PDF, etc.). O *SessionManager* pede então uma referência para o *TextResourceController* da sessão ao *Singleton ResourceManager* e invoca o seu novo método criado para esta contribuição, também intitulado *importText*. Tudo isto é feito no *SessionManager* dentro de um novo *thread*, de modo a não sobrecarregar o funcionamento da plataforma uma vez que o processo de conversão de documentos é geralmente pesado. Quanto ao método *exportText* do *SessionManager*, este recebe como argumento um *Text*, cria um novo *thread* e invoca o novo método *exportText* do *TextResourceController*.

No *WaveControllerBean*, o *Stateless* criado no capítulo 4.2 e que implementa a interface *TextResourceController*, o método *importText* começa por obter o documento por HTTP através do seu url. Depois o *Bean* cria uma nova Wave com os participantes da sessão, simbolizando a adição de um novo documento e adiciona-lhe o documento importado. Se o documento for do formato HTML, é usado directamente o *Parser* criado para efectuar a conversão. O outro formato para o qual foi desenvolvido o suporte para importação foi o DOCX. Para tal, usou-se a biblioteca docx4j [51]. Através desta era possível converter documentos DOCX para documentos

HTML. Uma vez nesse formato, usava-se o *Parser* criado para converter em Wave e submeter no servidor. Como o ficheiro DOCX era todo ele convertido num único ficheiro HTML e não havia forma no docx4j de criar vários ficheiros HTML, como um para cada página, o ficheiro DOCX era importado num único documento Wave.

O método *exportText* no *WaveControllerBean* adquiria o *id* do documento a exportar, a partir da entidade *Text*, obtinha o documento Wave do servidor através do *WaveConnector* e convertia-o para HTML usando o *Parser* criado para o efeito. Se o formato de exportação pretendido fosse o HTML, o *WaveConnector* comunicava com o *SessionManager* e pedia-lhe para publicar na Web o ficheiro com o documento. Tal era feito através da funcionalidade de partilha de ficheiros da plataforma. O outro formato de exportação contemplado foi o PDF. Para converter os documentos de HTML para PDF usou-se a ferramenta xhtmlrenderer [52]. Uma vez convertidos para PDF, eram publicados na Web da mesma maneira que os ficheiros HTML. Uma alternativa que foi adicionada ao método *exportText* foi de exportar não apenas um documento de cada vez mas também uma Wave inteira. Para tal, o cliente passava na entidade *Text* o *id* da Wave, em vez de enviar o *id* do documento. Como o resultado da exportação tinha que continuar a ser um documento singular, para exportar uma Wave inteira para HTML separava-se os vários documentos HTML convertidos através de uma *tag* personalizada que não pertence à lista de *tags* do HTML. A *tag* escolhida foi *<puc />*. De modo a poder-se posteriormente voltar a importar o documento em HTML e ficar com a Wave estruturada como estava antes da exportação, foi adicionado o suporte para esta *tag* no método *importText*. Para exportar uma Wave inteira para PDF, usou-se uma funcionalidade da biblioteca xhtmlrenderer que convertia vários ficheiros HTML num só PDF, separando-os por páginas.

Por fim, tentou-se também exportar documentos no formato DOCX. Para tal, usou-se a ferramenta docx4j à mesma para converter os documentos HTML para DOCX. No entanto, como esta não era uma funcionalidade nativa da ferramenta, foi necessário implementá-la através de outras funcionalidades da mesma. Assim, usou-se as funcionalidades de conversão de documentos através de gramáticas. O resultado obtido preservava todo o texto do documento e a sua estrutura (parágrafos, etc.). Porém, os estilos e anotações (tipo de letra, tamanho, cor, etc.) não eram preservados e, como tal, optou-se por incluir esta funcionalidade na plataforma apenas para testes.

4.4. Modelo de Extensibilidade

No capítulo 3.4 foi apresentada a solução concebida para a contribuição do modelo de extensibilidade da plataforma. A solução consistia em adicionar um novo servidor de recursos à plataforma e em desenvolver o suporte do lado da mesma para a sua utilização. Por fim foi concebida uma solução de carregamento dinâmico de funcionalidades colaborativas para substituir o modelo estático da plataforma base.

4.4.1. Widget Engine da Plataforma

Como a solução concebida no capítulo 3.4 foi em muito parecida à concebida no capítulo 3.2, também as implementações das duas soluções foram bastante idênticas. Em primeiro lugar era necessário escolher um *Widget Engine* que implementasse a API das Gadgets do Google Wave [44]. A *Widget Engine* escolhida foi o Apache Wookie [53]. O Apache Wookie era uma *Widget Engine*, escrita na linguagem Java, e que implementava a especificação de Widgets W3C [42], assim como a API das Gadgets do Google Wave. O Apache Wookie permitia ainda a utilização de *widgets* OpenSocial através da sua integração com outro *Widget Engine* da Apache, o Apache Shindig [54]. No entanto, o grande interesse de usar o servidor Wookie provinha do facto de implementar a API das Gadgets Wave, possibilitando assim a utilização de *widgets* colaborativas. Para receber pedidos das aplicações Web o Wookie possuía uma interface REST. As operações disponibilizadas através desta API incluíam operações para criar instâncias de uma dada *widget*, adicionar participantes a uma instância previamente criada, etc.

Como foi visto no capítulo 3.2, uma vez escolhido o *Widget Engine* era necessário integrá-lo na plataforma. Para integrar o Wookie era necessário implementar na camada *Resource* os dois novos componentes concebidos, *WidgetConnector e WidgetController*. Antes de serem implementados foram alterados os seus nomes de modo a melhor representar o *Widget Engine* escolhido. Os componentes passaram então a intitular-se *WookieConnector* e *WookieController*, respectivamente.

```
if (addParticipantToWidgetInstance(instance, participant_id, participant_display_name, participant_thumbnail_url)){
    try {
        Registry registry = LocateRegistry.getRegistry(1599);
        WookieConnectorRMIRemote puc = (WookieConnectorRMIRemote)registry.lookup("/WookieConnector");
        puc.participantAdded(request.getParameter("shareddatakey"), participant_id);
    } catch (Exception e) {
        e.printStackTrace();
    }
    Notifier.notifyWidgets(session, instance, Notifier.PARTICIPANTS_UPDATED);
    return true;
```

Listagem 11 – Excerto da classe *ParticipantsController* do servidor Apache Wookie, com o código para enviar o evento de participante adicionado à plataforma

Seguindo o exemplo do capítulo 4.2, o *WookieConnector* foi implementado como um *Singleton* e o *WookieController* como um *Stateless*. No entanto, ao contrário do caso do Wave, para integrar o servidor Wookie na plataforma foi necessário alterar uma pequena parte do seu código. Esta necessidade surgiu porque originalmente não estavam contemplados o envio de eventos para as plataformas cliente do servidor. Porém, para este trabalho havia dois eventos cuja recepção era imprescindível. Estes eram os eventos de participante adicionado e participante removido. Para efectuar este envio acedeu-se ao código do servidor Wookie, mais especificamente à *Servlet* que continha a interface REST para interacção com os participantes de instâncias de *widgets*, e adicionou-se o código para enviar os eventos para a plataforma.

A listagem 11 representa o código adicionado à *Servlet ParticipantsController*, no método que adiciona um participante a uma *widget*. Como se pode ver na listagem, para comunicar com a plataforma é usado Java RMI [55]. Optou-se por usar este meio de comunicação por tanto o servidor como a plataforma estarem escritos em Java e por este ter geralmente melhor performance face a alternativas Web. Para usar Java RMI foi necessário definir o *WookieController* como um servidor de RMI. A listagem 12 representa o *Singleton WookieControllerBean*. Comparando-o com o *Singleton* da listagem 4 no capítulo 4.2, este implementa uma interface adicional e estende da classe *UnicastRemoteObject* [55]. Desta forma o *WookieControllerBean* é definido como um servidor RMI. A interface adicional, *WookieConnectorRMIRemote*, estende da interface *Remote* [55] e específica os métodos remotos que podem ser invocados por RMI. Esta interface contém dois métodos, *participantAdded* e *participantRemoved*, invocados para enviar os eventos de participante adicionado e removido, respectivamente.

```
@Service
@Management(WookieConnectorLocal.class)
@Local(WookieConnectorLocal.class)
public class WookieConnectorBean extends UnicastRemoteObject implements WookieConnectorLocal, WookieConnectorRMIRemote {

    @EJB
    private ResourceManagerLocal resourceManager;
```

Listagem 12 – Implementação do componente *WookieConnector*

A listagem 13 representa o *Stateless WookieController*. Comparando-o com o *Stateless* da listagem 5, este implementa uma interface diferente, *WidgetController*, que estende da mesma interface genérica *ResourceController* mas adiciona métodos específicos de recursos de tipo *Widget*. De resto, os dois *ResourceControllers* são praticamente idênticos. No entanto nos métodos da interface *ResourceController*, para além das operações sobre entidades da plataforma, que são idênticas em ambos, os dois *Controllers* diferem nas interacções que fazem com os seus servidores de recursos:

- *setResources*: neste método o *WookieController* comunica com o *WookieConnector* e pede-lhe instanciar uma nova *widget* do tipo pretendido.
- *newResourceClient*: neste método o *WookieController* pede ao *WookieConnector* para obter um url para o novo cliente se ligar à instância de *widget* criada. No entanto, o cliente só é adicionado como participante ao estado da *widget* se a sessão já estiver aberta. Assim garante-se que este não pode usar o url até que tal aconteça.
- *disconnect*: neste método o *WookieController* pede ao *WookieConnector* para remover um participante da instância de *widget* da sessão de recursos que representa.
- *processEvent*: idêntico ao método do *Stateless* da listagem 5.
- *updateStatus*: quando é mudado o estado da sessão para aberta, o *WookieController* pede ao *WookieConnector* para adicionar os participantes da sessão ao estado da sua *widget*, podendo estes assim usar os urls que lhes foram atribuídos. Como no servidor Wookie existem operações para bloquear e resumir o estado de uma *widget*, usam-se estas quando é pausada e resumida a

sessão. Por fim, quando é fechada a sessão, bloqueia-se o estado da *widget* e removem-se os participantes da mesma.

```
@Stateless
@Local(WidgetController.class)
public class WookieControllerBean extends ResourceControllerAbstract implements WidgetController {

    @EJB
    private WookieConnectorLocal wookieConnector;
```

Listagem 13 – Implementação do componente *WookieController*

4.4.2. Carregamento Dinâmico de Funcionalidades

Como foi visto no capítulo 3.4, para implementar esta contribuição foi adicionado ao modelo de dados uma nova entidade. A listagem 14 representa essa entidade, intitulado *ResourceFeature*. Como na entidade implementada no capítulo 4.3.1, esta é definida através das anotações *@Entity* e *@Table*. Adicionalmente, os atributos da entidade também possuem anotações para definir o nome e tamanho das suas colunas na base de dados.

Para os clientes poderem obter as instâncias da entidade presentes na base de dados e para as mesmas poderem ser persistidas, foram adicionados métodos ao *Stateless ResourceService*. Este é o *Bean* usado para obter e guardar na base de dados as entidades ligadas aos recursos, como por exemplo a *ResourceSession*. Assim o *ResourceService* passou também a ser responsável por obter e persistir o *ResourceFeature*. Para além do método normal de obter entidades pelo seu identificador, foi adicionado um método que obtinha uma *ResourceFeature* pelo seu atributo *featureId*. Este é o atributo que identifica unicamente uma funcionalidade. Um exemplo de valor é "Document Edition". Chegou a considerar-se se este atributo não poderia ser usado pela base de dados para identificar unicamente a entidade, descartando assim a necessidade da existência do atributo *id*. Porém, optou-se por não o fazer pois usar uma String como chave primária da tabela de uma base de dados relacional seria sempre mais custoso, em termos de performance, que usar um valor numérico.

```java
@Entity
@Table(name = "resourceFeature")
public class ResourceFeature implements Serializable{

    @OneToOne(mappedBy="resourceFeature")
    private ResourceSessionDescription resourceSessionDescription;

    @Id
    @Column(name = "id")
    @GeneratedValue
    private Integer id;

    @Column(name = "featureId", length = 100)
    private String featureId;

    @Column(name = "title", length = 100)
    private String title;

    @Column(name = "description", length = 100)
    private String description;

    @Column(name = "iconURL", length = 100)
    private String iconURL;
```

Listagem 14 – Implementação do componente *ResourceFeature*

Uma vez que os clientes não podiam contactar directamente o *ResourceService*, foram adicionados métodos para este propósito ao *Singleton* que o usava. Este era o *ResourceManager*. Os métodos adicionados foram *getResourceFeatures*, que devolvia o conjunto de todas as *ResourceFeatures* presentes na base de dados, e *getResourceFeature*, que obtinha uma *ResourceFeature* pelo seu *featureId*.

Por fim, foi implementado no *ResourceManager* um mecanismo para consultar periodicamente os servidores de recursos e actualizar a tabela de *ResourceFeatures* da base de dados. Como o servidor Wookie era, no momento de implementação deste trabalho, o único servidor de recursos cujas funcionalidades poderiam ser modificadas dinamicamente, o mecanismo foi implementado só para o seu caso.

Para obter as novas funcionalidades do servidor Wookie, o *ResourceManager* comunicava com o *Stateless WookieManager*, através da sua interface local, e pedia-lhe o seu conjunto de funcionalidades. Por sua vez, o *WookieManager* contactava o *WookieConnector* e pedia-lhe para aceder ao servidor Wookie e adquirir o seu conjunto de *widgets*. Cada *widget* devolvida corresponderia a uma funcionalidade diferente para a plataforma, ou seja, ao receber o conjunto de *widgets* disponíveis no servidor *Wookie*, o *ResourceManager* verificava quais é que eram novas e criava uma *ResourceFeature* para cada uma. De modo a automatizar este processo, o método criado no *ResourceManager* para efectuar esta operação começava por lançar um *thread* novo e iniciar um ciclo infinito. Neste ciclo o *ResourceManager* consultava o *WookieManager*, efectuava a sua lógica e depois suspendia a execução do *thread* por um determinado tempo configurável. O tempo ideal pensado foi de 24 horas, ou seja, num ambiente de distribuição da

plataforma em que esta operaria sem interrupções, o processo de sincronização de funcionalidades ocorreria uma vez por dia.

5. Validação

Neste capítulo serão apresentadas as metodologias escolhidas para validar as contribuições concebidas e implementadas nesta dissertação. As metodologias usadas dividiram-se em três famílias distintas: testes de portabilidade, testes de performance e testes de usabilidade. A primeira vertente foi usada apenas para validar a primeira contribuição concebida, a interface portável para o Wave. As outras duas vertentes foram aplicadas a todas as contribuições implementadas. Para cada processo de validação efectuado é explicado (1) qual o seu objectivo e o que se pretendia validar, (2) em que consistiu o teste realizado, (3) condições e ambiente em que foi executado e (4) análise de resultados e conclusões. No fim do capítulo é feita uma discussão final geral sobre os diferentes resultados obtidos.

5.1. Testes de Portabilidade

No caso da contribuição da interface portável para o Wave, como o grande objectivo da mesma era atingir o maior nível de portabilidade possível era imprescindível validar os resultados conseguidos. Este tipo de validação apenas foi feito para esta contribuição pois ela era a única que tinha como objectivo ser usada directamente em dispositivos heterogéneos.

1. **Objectivo do teste**

Validar a portabilidade da interface concebida.

2. **Descrição do teste**

O teste consistiu em executar um cliente Wave em diferentes terminais, incluindo dispositivos móveis. O cliente comunicaria com o servidor Wave através da interface implementada.

3. **Ambiente de realização do teste**

Para efectuar este teste foi criado um cliente Wave que usava a interface concebida. O cliente foi construído a partir de uma versão inicial de um protótipo desenvolvido no âmbito de outra dissertação, [41]. Essa dissertação ocorreu na PT Inovação em paralelo com esta.

O cliente foi implementado em Javascript e HTML, através da *Framework* GWT [73]. Para validar a portabilidade da interface o cliente foi executado (1) nos *browsers* para PC Internet

Explorer 8 (versão 8.0.7600.16385), Google Chrome (versão 5.0.375.99) e Safari, (2) num *smart phone* HTC Hero com sistema operativo Android 2.1 e (3) num IPod Touch. Para cada terminal listado testou-se o cliente em duas situações. A primeira foi com o cliente instalado no servidor JBoss (5.1.0). A segunda foi com o cliente instalado localmente nos dispositivos. No caso dos dispositivos móveis foi necessário compilar o cliente para poder executar localmente nos mesmos.

Em todos os ambientes testados, a interface portável foi instalada no servidor Web Jetty 7.0.1.

4. Resultados do teste

O cliente Wave que usava a interface funcionou correctamente em todos os ambientes testados, tendo sido possível tanto efectuar operações como receber actualizações assincronamente. A secção 1 dos anexos mostra o cliente criado a funcionar nos diversos ambientes.

5.2. Testes de Performance

Para validar a performance das contribuições desenvolvidas foram efectuados testes de carga sob os componentes implementados em cada uma. Dependendo da contribuição, os testes de carga foram efectuados em condições e ambientes diferentes. Este capítulo divide-se portanto em quatro subcapítulos, um para cada grande contribuição desenvolvida.

5.2.1. Interface Portável para o Wave

Para testar a performance da interface portável para o Wave implementada no capítulo 4.1 foram efectuados dois testes. Um validou a performance da interface com múltiplos utilizadores e a outra validou a performance de uma das operações da interface e comparou-a com o uso directo do servidor Wave.

5.2.1.1. Teste com Múltiplos Utilizadores

1. Objectivo do teste

Validar a performance da interface portável com acessos concorrentes de múltiplos utilizadores.

2. Descrição do teste

O teste consistiu em lançar mil *threads* concorrentemente e efectuar a operação de *login* na interface com cada um. Ao mesmo tempo foram registados os tempos de execução das

operações. Para aumentar o paralelismo do teste, foram usados dois PCs e os *threads* foram divididos pelos dois, ou seja, em cada PC foram lançados 500 *threads*. O teste foi repetido cem vezes, de modo a poder-se efectuar uma análise mais detalhada e de modo a filtrar possíveis irregularidades.

3. Ambiente de realização do teste

A interface portável foi instalada no servidor Web Jetty 7.0.1, a correr num PC com processador Intel Pentium D 3.41Ghz, 2 GB de RAM, 160 GB de disco e sistema operativo Windows 7 32-bit.

O servidor Wave FedOne foi instalado numa máquina virtual com processador Intel Xeon 2.13 Ghz Dual Core, 2 GB de RAM, 10 GB de disco e sistema operativo Linux Redhat 4. Para realizar este teste aumentou-se o valor do Java *Heap Space* da máquina virtual criada para correr o servidor FedOne para um máximo de 1 GB. A comunicação entre os dois servidores foi feita através da rede interna da PT Inovação, com velocidade máxima de transmissão de 100 MB.

Os *threads* que simulavam o comportamento de um cliente da plataforma foram lançados em dois PCs adicionais. O PC 1 tinha processador Intel Core 2 Duo P8600 de 2.4 GHz, 3 GB de RAM, 300 GB de disco e sistema operativo Windows 7 32-bit. O PC 2 tinha processador Intel Core 2 Duo E4400 de 2 GHz, 2 GB de RAM, 250 GB de disco e sistema operativo Windows Vista 32-bit.

4. Resultados do teste

A figura 14 mostra um gráfico representativo dos resultados alcançados nas cem execuções do teste. O eixo do x representa os clientes lançados em cada PC e o eixo do y representa o tempo em milissegundos. No gráfico são visíveis duas linhas tendência de média móvel com períodos de 50 unidades, construídas a partir dos dados obtidos dos dois PCs.

Da análise do gráfico conclui-se que ao longo do tempo as medições mantêm-se estáveis dentro de um intervalo de aproximadamente 60 milissegundos, ou seja, mesmo com o aumento progressivo de carga a interface portável continua a mostrar um desempenho constante. Por outro lado, nota-se alguma irregularidade entre medições, com picos altos seguidos de picos baixos. No entanto, ao comparar os gráficos das diferentes execuções feitas não foi possível observar qualquer padrão. A conclusão que se tira deste resultado é que os diferentes picos são consequência da sobrecarga da rede interna da PT Inovação. Tentou-se realizar algumas das execuções do teste fora do horário normal de trabalho, porém mesmo nessas condições os resultados mostravam sempre alguma variância.

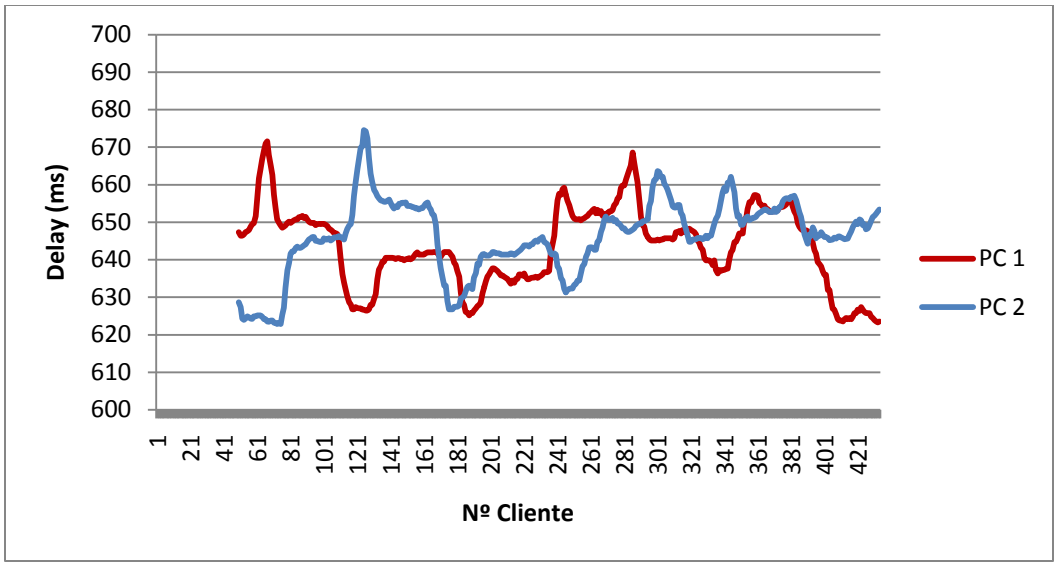


Figura 14 – Múltiplos utilizadores na interface portável - Gráfico com médias móveis de 50 períodos

5.2.1.2. Teste Comparativo com Acesso Directo ao Wave

1. Objectivo do teste

Validar a performance de um dos métodos da interface contra o desempenho obtido de aceder directamente ao servidor Wave.

2. Descrição do teste

O teste consistiu em invocar um dos métodos da interface portável e em efectuar a mesma operação contactando directamente o servidor Wave, repetindo ambos os procedimentos mil vezes e medindo os tempos de cada um. A operação escolhida foi a adição de um novo Blip no fim de uma Wave. O teste foi repetido cem vezes, de modo a poder-se efectuar uma análise mais detalhada e de modo a filtrar possíveis irregularidades.

3. Ambiente de realização do teste

O servidor Wave e a interface portável foram instalados como descrito no capítulo 5.2.1.1. A execução de ambos os testes foi feita a partir do mesmo PC onde foi instalado o servidor Jetty com a interface.

4. Resultados do teste

A figura 15 é um gráfico representativo dos resultados obtidos nas cem execuções deste teste. No eixo do x é registado o número do Blip criado e no eixo do y é registado o tempo que demorou a criar cada Blip. O gráfico mostra duas linhas tendência, uma para o teste realizado

directamente sobre o Wave e uma para o teste da interface. As linhas foram construídas a partir de uma média móvel com períodos de 100 unidades.

O gráfico mostra que o uso da interface portável leva a um acréscimo de latência médio na ordem dos 40 milissegundos. Pensa-se que mesmo num sistema em tempo quase real como o Wave a latência adicional não é significativa, validando assim a performance da interface desenvolvida. Uma análise mais pormenorizada do gráfico mostra um pequeno aumento de latência ao longo do tempo. No entanto este aumento é muito pouco significativo e verifica-se em ambas as linhas, ou seja, provém do servidor Wave. É também observável que as medições de tempos são mais instáveis na interface portável face ao uso directo do Wave. No entanto não foi notado qualquer padrão nas diferentes medições. A conclusão feita destes dados é que a variância nas medições é consequência de tráfico na rede interna da PT Inovação ou sobrecarga de processos a correr em paralelo no PC de testes.

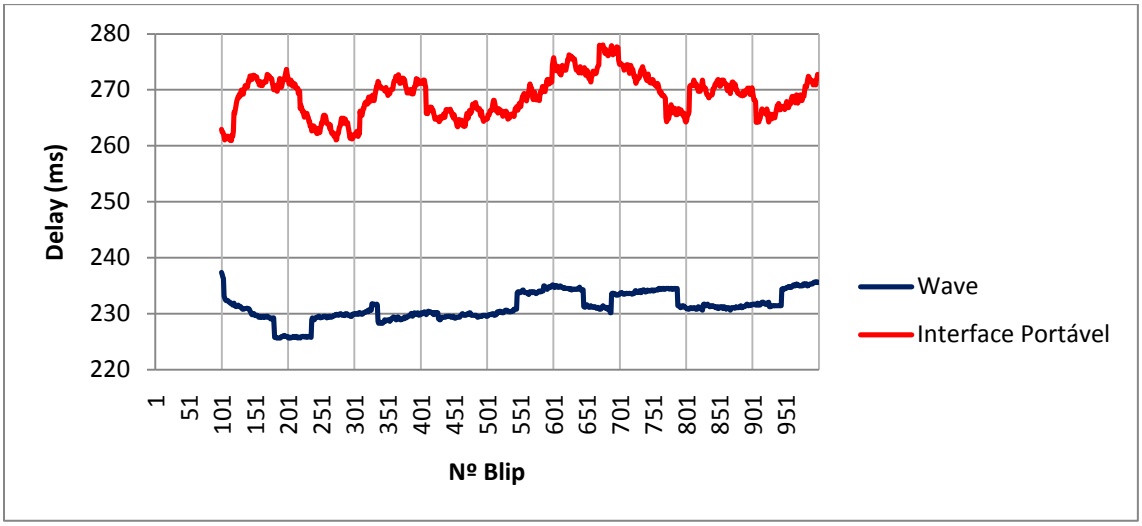


Figura 15 – Teste comparativo com acesso directo - Gráfico com médias móveis de 100 períodos

5.2.2. Integração do Wave na Plataforma

1. Objectivo do teste

Validar o desempenho temporal dos módulos desenvolvidos para integrar o servidor Wave FedOne na plataforma. Comparar os resultados obtidos com a performance dos outros componentes da plataforma na realização do mesmo teste.

2. Descrição do teste

O teste consistiu em efectuar, concorrentemente, mil sessões colaborativas da plataforma com a funcionalidade colaborativa *Document Edition* e o servidor de recursos Wave. Por

efectuar uma sessão entende-se todo o processo de construção da mesma: criação de um grupo, em que cada grupo criado possuía três participantes; criação de uma conversa com esse grupo; criação da sessão a partir da conversa; efectuar a reserva de recursos da sessão através do *setup* da mesma.

3. Ambiente de realização do teste

Para realizar o teste foi instalada a plataforma no servidor aplicacional JBoss 5.1.0. O servidor JBoss foi instalado num PC com processador Intel Pentium D 3.41Ghz, 2 GB de RAM, 160 GB de disco rígido e sistema operativo Windows 7 32-bit. O JBoss foi executado a partir do IDE Eclipse, através do plugin JBoss Tools [75]. O teste foi implementado através do serviço *Session Scheduler* do JBoss [71]. Para medir a performance temporal da plataforma usou-se uma ferramenta para *profiling* do servidor JBoss no Eclipse, o *Eclipse Test & Performance Tools Platform* [74].

O servidor Wave FedOne foi instalado numa máquina virtual com processador Intel Xeon 2.13 Ghz Dual Core, 256 MB de RAM, 10 GB de disco e sistema operativo Linux Redhat 4. A comunicação entre os dois servidores foi feita através da rede interna da PT Inovação, com velocidade máxima de transmissão de 100 MB.

Como modo de persistência, a plataforma usou uma base de dados relacional PostgreSQL [72], instalada num servidor com processador Intel Xeon Dual-Core com 2.13 Ghz, 4 GB de RAM e 160 GB de disco. A comunicação entre a plataforma e a base de dados também foi feita pela rede interna da PT Inovação.

4. Resultados do teste

O relatório gerado pela ferramenta de *profiling* do JBoss foi guardado como relatório de texto. Este continha os dados brutos agrupados por classes. Os dados estavam separados por colunas, havendo quatro colunas diferentes, ou seja, havia quatro tipos de medições feitas. A primeira coluna, *Average Base Time*, representava o tempo médio que a invocação de cada classe demorava. A coluna *Calls* representava o número de invocações feitas a cada classe e a coluna *Base Time* representava o tempo total gasto em cada classe, ou seja, o tempo médio vezes o número de invocações.

A partir do relatório de texto foram criados gráficos para analisar mais detalhadamente a performance de cada componente da plataforma. A figura 16 representa um gráfico circular com o tempo total gasto em cada classe. Como se pode observar, 52% do tempo de realização do teste foi gasto na classe *GenericDAOImpl*. Esta é a classe que faz a ligação entre a plataforma e a base de dados. O tempo representa apenas operações de gravação de dados, pois no teste realizado não eram feitas consultas à base de dados. A conclusão feita sobre estes resultados é que o complexo modelo de dados da plataforma tem um elevado custo quando é necessário persistir dados.

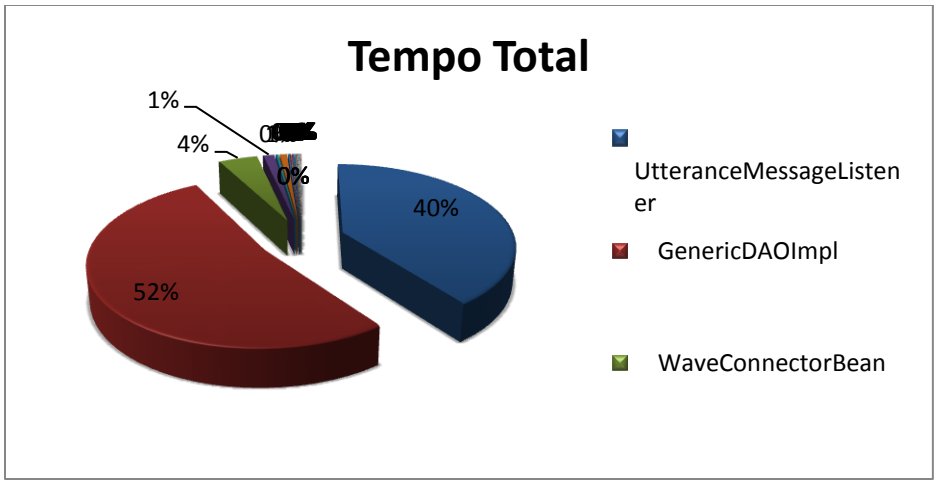


Figura 16 – Integração do Wave na Plataforma - Tempo total gasto em cada classe

A classe *UtteranceMessageListener*, apresentada no capítulo 4.3.2, corresponde a 40% do tempo total gasto. Uma análise da performance de cada método mostrou que 100% deste tempo é gasto no construtor da classe. Neste construtor é instanciado um *MessageConsumer* de JMS e é feito o *lookup* manual de um *Bean*, o *UtteranceManager*. A criação do *MessageConsumer* é necessária porque não se implementou este componente como um *Message-Driven Bean* e o *lookup* é necessário exactamente por a classe não ser um *Bean*. Não foi possível usar o *Message-Driven Bean* por serem criados tópicos dinamicamente. Concluindo, o tempo registado neste componente é consequência da criação de tópicos dinamicamente.

A terceira classe com mais tempo gasto é o componente desenvolvido nesta contribuição, o *WaveConnector*. No entanto, este componente apenas representa 4% do tempo total gasto. Uma análise da performance dos métodos desta classe mostra que 100% do seu tempo é dispendido no método *newWave*, método que acede ao servidor Wave e cria uma Wave nova. Comparando o tempo médio de execução deste método face aos resultados da validação feita à performance do servidor Wave no capítulo 5.2.1.2, concluí-se que a maior parte do tempo dispendido neste método é gasto a contactar o servidor Wave.

Uma conclusão final dos resultados deste teste é que os componentes desenvolvidos para esta contribuição demonstram um nível de performance bastante elevado comparativamente com a performance dos outros componentes da plataforma.

5.2.3. Funcionalidades de Recording e Play

1. Objectivo do teste

Validar o desempenho temporal dos módulos desenvolvidos para as funcionalidades de *Recording* e *Play*.

2. Descrição do teste

O teste consistiu em simular uma sessão colaborativa com o recurso s*lideshow* e efectuar o *recording* da mesma. O *recording* foi feito através do *UtteranceGuide* slide, ou seja, o *recording* das *utterances* foi guiado por transição de slide. Foi também usada a funcionalidade de sincronização com recursos de texto, através do servidor Wave. O *slideshow* usado continha dez slides e o teste foi executado cem vezes, ou seja, no total foram registados 1000 slides.

3. Ambiente de realização do teste

O ambiente de execução deste teste foi o mesmo descrito no capítulo 5.2.2.

4. Resultados do teste

O relatório gerado pela ferramenta de *profiling* do JBoss foi guardado como ficheiro de texto. A partir do relatório de texto foi criado um gráfico circular com percentagens dos tempos dispendidos em cada classe. A figura 17 representa esse gráfico.

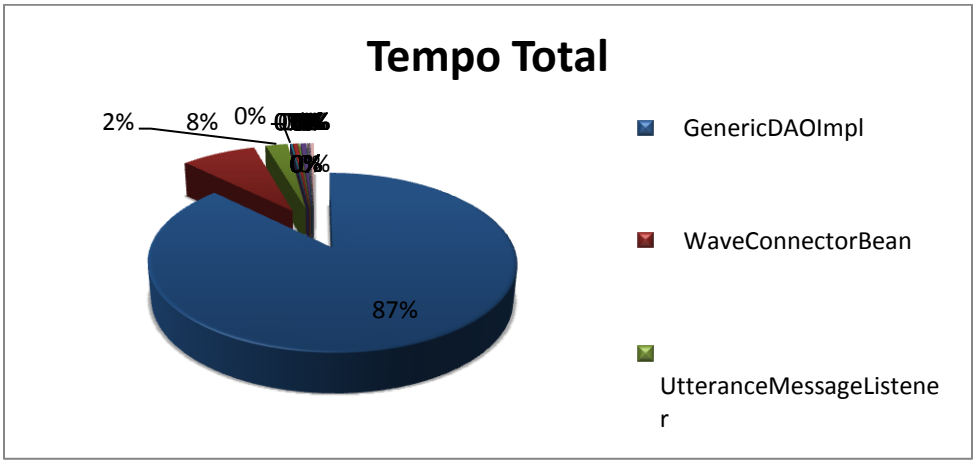


Figura 17 – Funcionalidades de Recording e Play - Tempo total gasto em cada classe

Pelo gráfico da figura 17 pode-se novamente concluir que o grande *bottleneck* da plataforma é o acesso à base de dados, representando neste teste 87% do tempo total gasto. Isto é consequência do seu complexo modelo de dados, herdado da plataforma base. Pode-se argumentar que ao juntar-se uma nova entidade ao modelo de dados (o *Session Document*) está-se a contribuir negativamente para este *bottleneck*, porém a criação da entidade era estritamente necessária de modo implementar esta contribuição. Não obstante existem outras formas de implementar a entidade, incluindo a anotação *@Embeddable* [76], que deverão ser analisadas num trabalho futuro.

O segundo maior componente do gráfico, apesar de apenas representar 8% do seu total, é o *WaveConnector*. Este resultado é consequência da sincronização feita com o recurso de texto no

fim de cada *utterance*. No entanto os tempos medidos não deixam de ser naturais quando comparados com os registados no capítulo 5.2.1.2.

Por fim concluí-se que os componentes desenvolvidos para esta contribuição, incluindo o *UtteranceManagerBean*, *UtteranceMessageListener* e o *BrokerConnector*, apresentam um nível de performance bastante bom uma vez que são das classes mais invocadas no teste realizado e ainda assim apresentam um tempo médio e total bastante baixos.

5.2.4. Modelo de Extensibilidade

1. **Objectivo do teste**

Validar o desempenho temporal dos módulos desenvolvidos para integrar o *Widget Engine* Wookie na plataforma. Comparar os resultados obtidos com a performance dos outros componentes da plataforma na realização do mesmo teste.

2. **Descrição do teste**

O teste consistiu em efectuar, concorrentemente, mil sessões colaborativas da plataforma com a funcionalidade colaborativa *widget* e o servidor de recursos Wookie. Por efectuar uma sessão entende-se todo o processo de construção da mesma: criação de um grupo, em que cada grupo criado possuía três participantes; criação de uma conversa com esse grupo; criação da sessão a partir da conversa; efectuar a reserva de recursos da sessão através do *setup* da mesma. Para seleccionar a funcionalidade colaborativa da sessão usou-se o sistema dinâmico de carregamento de funcionalidades desenvolvido. Mais especificamente, usou-se uma das *widgets* que vinha já instalada no servidor Wookie, denominada de *Natter*.

3. **Ambiente de realização do teste**

O ambiente de execução deste teste foi idêntico ao descrito no capítulo 5.2.2. No entanto, o servidor de recursos usado não foi o Wave, mas sim o Wookie. A instalação do Wookie foi feita a partir do seu arquivo WAR, *deployed* na mesma instância do servidor JBoss onde estava a ser executada a plataforma.

4. **Resultados do teste**

O relatório gerado pela ferramenta de *profiling* do JBoss foi guardado como ficheiro de texto. A partir do relatório de texto foi criado um gráfico circular com percentagens dos tempos dispendidos em cada classe. A figura 18 representa esse gráfico.

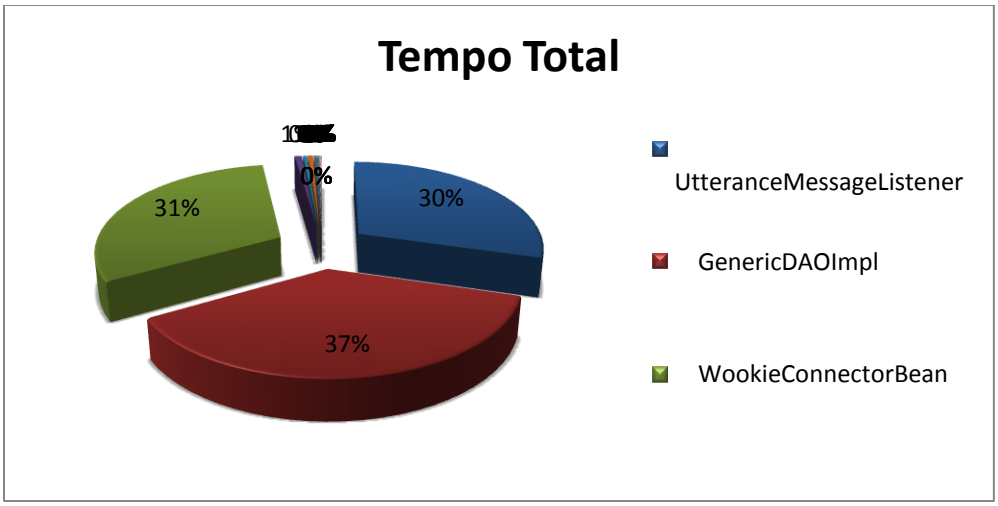


Figura 18 – Modelo de Extensibilidade - Tempo total gasto em cada classe

Como se pode ver pela figura, o grande *bottleneck* da plataforma continua a ser o acesso à base de dados, com 37% do tempo gasto. O *UtteranceMessageListener* e o *WookieConnector* estão quase empatados em 2º lugar, com respectivamente 30% e 31% do tempo total gasto. Comparando com os resultados do capítulo 5.2.2, as classes *GenericDAOImpl* e *UtteranceMessageListener* têm aproximadamente os mesmos tempos. No entanto, nota-se que é dispendido bastante mais tempo no *WookieConnector* do que no *WaveConnector*. Um teste de performance feito directamente ao servidor Wookie mostrou que esse acréscimo de tempo gasto provém da comunicação com o servidor.

A conclusão deste resultado é que a API do Wookie não tem tão bom desempenho como a do Wave. Isto acontece porque enquanto no Wave a comunicação é feita através do protocolo cliente-servidor do FedOne (protocolo RPC com protobufs sobre TCP), no Wookie a comunicação é feita por HTTP. Por outro lado, enquanto no Wave o processo de *setup* apenas comunica uma vez com o servidor, para criar a Wave, no caso do Wookie a plataforma tem que comunicar com o servidor para instanciar a *widget* escolhida e, para cada participante da sessão, volta a ter que comunicar com o servidor para obter o url ao qual ele se vai ligar. Como neste teste cada sessão continha três participantes e foram criadas mil sessões, o servidor Wookie foi contactado quatro mil vezes.

5.3. Testes de Performance após Optimizações

Após as optimizações que foram realizadas no sistema de eventos da plataforma (ver cap. 4.3.4), foi novamente avaliado o desempenho dos vários componentes da plataforma, de forma a validar o resultado das optimizações efectuadas. Para tal, foram repetidos os mesmos tipos de testes, tendo sido aproveitada a oportunidade para aumentar a escala dos mesmos.

5.3.1. Integração do Wave na Plataforma

1. Objectivo do teste

Validar o desempenho temporal dos módulos da plataforma na criação de sessões com o recurso Wave. Comparar os resultados obtidos com os analisados no capítulo 5.2.2.

2. Descrição do teste

O teste consistiu em efectuar, concorrentemente, três mil sessões colaborativas da plataforma com a funcionalidade colaborativa *Document Edition* e o servidor de recursos Wave. Por efectuar uma sessão entende-se todo o processo de construção da mesma: criação de um grupo, em que cada grupo criado possuía três participantes; criação de uma conversa com esse grupo; criação da sessão a partir da conversa; efectuar a reserva de recursos da sessão através do *setup* da mesma. Comparando com a validação original, neste teste foram efectuadas mais duas mil sessões concorrentemente. Pretendia-se elevar a carga para as dez mil sessões concorrentes, porém tal não foi possível devido às limitações impostas pela máquina disponível para realizar os testes.

3. Ambiente de realização do teste

Para realizar o teste foi instalada a plataforma no servidor aplicacional JBoss 5.1.0. O servidor JBoss foi instalado num PC com processador Intel Core 2 Duo 2.40Ghz, 3 GB de RAM, 320 GB de disco rígido e sistema operativo Windows 7 32-bit. O JBoss foi executado a partir do IDE Eclipse, através do plugin JBoss Tools [75]. O teste foi implementado através do serviço *Session Scheduler* do JBoss [71]. Para medir a performance temporal da plataforma usou-se uma ferramenta para *profiling* do servidor JBoss no Eclipse, o *Eclipse Test & Performance Tools Platform* [74].

O servidor Wave FedOne e a base de dados da plataforma foram instaladas nas mesmas condições descritas no capítulo 5.2.2.

4. Resultados do teste

A figura 19 representa o tempo total gasto em cada componente da plataforma para realizar o teste. A primeira observação que é feita a partir da figura é que o componente *UtteranceMessageListener*, que no capítulo 5.2.2 correspondia a 40% do tempo gasto e que foi um dos alvos das optimizações feitas, deixou de aparecer no gráfico. Analisando os dados brutos obtidos, confirmou-se que o componente já não é usado no processo de criação e setup de sessões. Por outro lado, o componente *GenericDAOImpl* continua a ter a maior percentagem de tempo gasto, 66%. No gráfico é também visível um novo componente, *SessionServiceBean*. Este componente faz a ligação entre o *GenericDAOImpl* e os restantes módulos da plataforma. Por outras palavras, este componente também está ligado ao acesso à base de dados. Nos testes originais este componente não aparecia porque os dados da sessão eram todos obtidos de forma

"*Eager*", ou seja, todos de uma só vez e tal era feito no *GenericDAOImpl*. Quando os novos testes foram realizados, este acesso à base de dados tinha sofrido optimizações neste campo, realizadas por outros membros da equipa de desenvolvimento da plataforma. Nomeadamente, alguns dos atributos da sessão colaborativa eram obtidos da base de dados de forma "*Lazy*", ou seja, só eram obtidos quando tal era necessário. No entanto, por motivos de teste, eles estavam a ser forçadamente obtidos no *SessionServiceBean* de forma a garantir que estavam correctos.

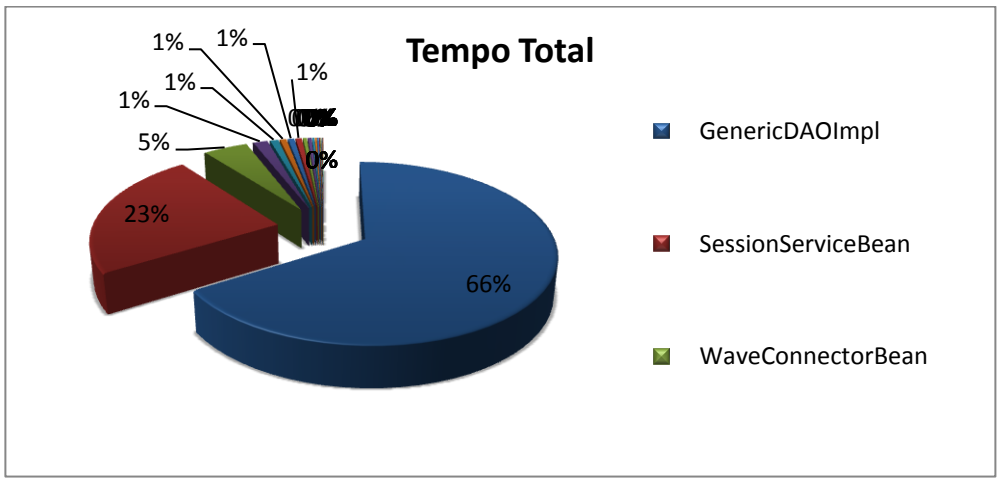


Figura 19 - Integração do Wave na Plataforma - Tempo total gasto em cada classe após optimizações

Resumindo, ambos os componentes *GenericDAOImpl* e *SessionServiceBean* correspondem a acessos de base de dados, o que significa que com as optimizações desenvolvidas sobre o sistema de eventos, 89% do tempo gasto na criação e setup de sessões com o Wave é gasto com a base de dados.

O componente *WaveConnectorBean* corresponde a 5% do tempo gasto, o que confirma as conclusões tiradas no parágrafo anterior uma vez que este componete quase não sofreu alteração quando comparado com as validações originais (4%).

5.3.2. Funcionalidades de Recording e Play

1. Objectivo do teste

Validar o desempenho temporal dos módulos desenvolvidos para as funcionalidades de *Recording* e *Play* após as optimizações efectuadas ao sistema de eventos. Comparar os resultados obtidos com os das validações originais.

2. Descrição do teste

O teste consistiu em simular uma sessão colaborativa com o recurso s*lideshow* e efectuar o *recording* da mesma. O *recording* foi feito através do *UtteranceGuide* slide, ou seja, o *recording*

das *utterances* foi guiado por transição de slide. Foi também usada a funcionalidade de sincronização com recursos de texto, através do servidor Wave. O *slideshow* usado continha 100 slides e o teste foi executado 50 vezes, ou seja, no total foram registados 5000 slides. Comparando com a validação original, foram enviados mais 4000 slides. Pretendia-se elevar a carga para os dez mil slides, no entanto tal não foi possível devido às limitações da máquina disponível para realizar os testes.

3. Ambiente de realização do teste

O ambiente de execução deste teste foi o mesmo descrito no capítulo 5.3.1.

4. Resultados do teste

A figura 20 representa o gráfico obtido a partir dos resultados do teste efectuado. Analisando o gráfico, nota-se que não são visíveis os componentes que foram alvo de optimizações (*UtteranceMessageListener*, *BrokerConnectorBean*, etc.), ou seja, as optimizações desenvolvidas não tiveram efeitos secundários negativos. Comparando com os resultados das validações originais (cap. 5.2.3), nota-se no entanto um aumento da percentagem do *WaveConnectorBean* e diminuição do tempo gasto com a base de dados (*GenericDAOImpl* e *SessionServiceBean*). Isto é resultado das optimizações desenvolvidas na base de dados por outros elementos da equipa de desenvolvimento da plataforma.

Resumindo, a conclusão que se pode retirar deste teste é que as optimizações desenvolvidas sobre o sistema de eventos, cujo principal resultado foi optimizar o tempo de criação e setup de sessões, como foi visto no capítulo anterior, não teve efeitos secundários negativos aquando da utilização do mesmo para *Recording* de sessões.

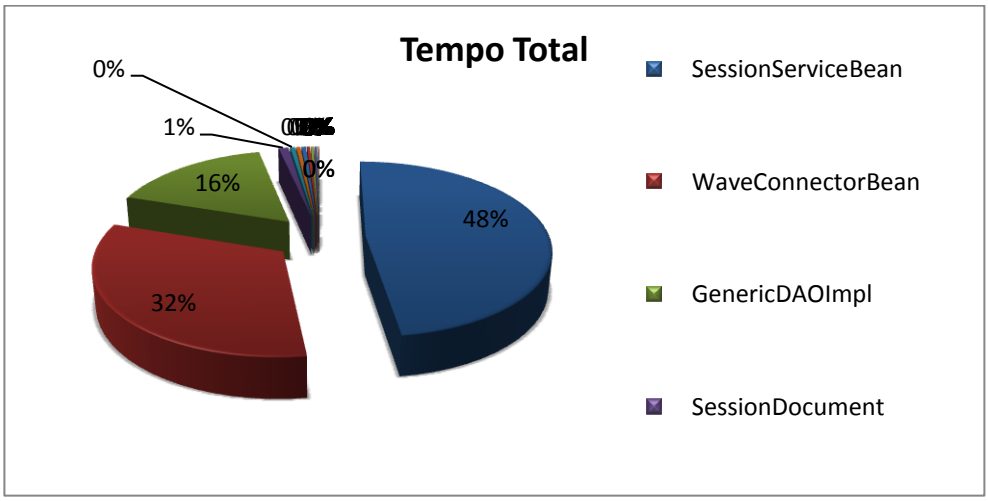


Figura 20 - Funcionalidades de Recording e Play - Tempo total gasto em cada classe após optimizações

5.3.3. Modelo de Extensibilidade

1. Objectivo do teste

Validar o desempenho temporal dos módulos da plataforma na criação de sessões com o recurso Wookie [53]. Comparar os resultados obtidos com os analisados no capítulo 5.2.4.

2. Descrição do teste

O teste consistiu em efectuar, concorrentemente, três mil sessões colaborativas da plataforma com a funcionalidade colaborativa *widget* e o servidor de recursos Wookie. Por efectuar uma sessão entende-se todo o processo de construção da mesma: criação de um grupo, em que cada grupo criado possuía três participantes; criação de uma conversa com esse grupo; criação da sessão a partir da conversa; efectuar a reserva de recursos da sessão através do *setup* da mesma. Para seleccionar a funcionalidade colaborativa da sessão usou-se o sistema dinâmico de carregamento de funcionalidades desenvolvido. Mais especificamente, usou-se uma das *widgets* que vinha já instalada no servidor Wookie, denominada de *Natter*.

3. Ambiente de realização do teste

O ambiente de execução deste teste foi idêntico ao descrito no capítulo 5.3.1. No entanto, o servidor de recursos usado não foi o Wave, mas sim o Wookie. O servidor Wookie foi instalado num servidor aplicacional JBoss, que por sua vez estava instalado num servidor com processador Intel Xeon Dual-Core com 2.13 Ghz, 4 GB de RAM e 160 GB de disco. A base de dados do servidor Wookie, em MySql, foi instalada na mesma máquina.

4. Resultados do teste

A figura 21 representa o gráfico com os tempos totais gastos por componente, obtidos a partir do teste efectuado. Tal como no capítulo 5.3.1, ao analisar o gráfico reparamos que não é visível o componente *UtteranceMessageListener*, componente esse que no capítulo 5.2.4 correspondia a 30% do tempo gasto. Esta análise confirma as conclusões retiradas no capítulo 5.3.1 sobre o sucesso das optimizações efectuadas sobre o sistema de eventos.

Quanto aos outros componentes visíveis no gráfico, nota-se que houve uma diminuição quanto ao tempo gasto no *WookieConnectorBean*, de 31% para 19%. Tal deve-se à deslocação do servidor Wookie e sua base de dados MySql para uma máquina diferente da que estava a fazer os testes de carga. Desta forma o servidor Wookie passou a ter mais recursos disponíveis, uma vez que os testes de carga e *profiling* da plataforma consumem bastantes recursos. Juntando as percentagens dos dois componentes ligados à base de dados, conclui-se que na criação e setup de sessões com o recurso Wookie, 73% do tempo gasto corresponde à base de dados.

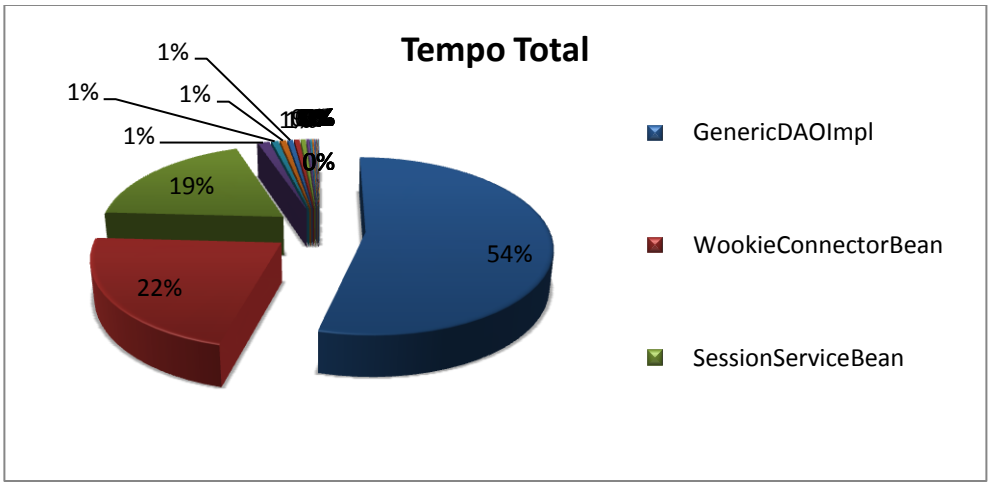


Figura 21 - Modelo de Extensibilidade - Tempo total gasto em cada classe após optimizações

5.4. Testes de Usabilidade

Para validar a usabilidade das contribuições desenvolvidas foram criados questionários de usabilidade. Estes questionários foram usados para entrevistar os potenciais clientes que iriam usar as contribuições implementadas. Mais especificamente, foram criados dois questionários diferentes, pois os possíveis utilizadores das contribuições concebidas agrupavam-se em dois grupos distintos. O primeiro questionário dirigiu-se aos utilizadores do Wave e do servidor FedOne, enquanto o segundo questionário dirigiu-se aos utilizadores da plataforma.

5.4.1. Interface Portável para o Wave

1. Objectivo do teste

Validar a usabilidade da interface portável concebida, assim como recolher dados sobre a aceitação geral do trabalho realizado nesta contribuição.

2. Descrição do teste

O teste consistiu na realização de um questionário sobre a interface portável para o Wave. O questionário foi dividido em três partes: primeiro foram feitas algumas perguntas sobre a experiência do questionado na área de colaboração e familiaridade com as tecnologias e protocolos Wave, de modo a aferir o seu *background*. Depois foi avaliada a satisfação com o actual protocolo cliente-servidor e com o *console client* do projecto FedOne, em termos de performance e portabilidade. Por fim, foi feita uma apresentação sobre o trabalho desenvolvido e foi avaliado o interesse no mesmo e nos seus resultados.

3. Ambiente de realização do teste

A população alvo deste questionário foi o grupo de discussão dos protocolos Wave e do projecto FedOne (grupo "Wave Protocol" [80]). Como tal o questionário foi criado através do serviço Google Docs, assim como a apresentação sobre a interface. A partir deste serviço foi disponibilizada uma versão online do questionário e da apresentação que foi anunciada ao grupo referido. Adicionalmente, foi questionado o autor de [41], pois este teve oportunidade de usar a interface no desenvolvimento da sua tese.

4. Resultados do teste

A secção 2 dos anexos mostra um resumo dos resultados obtidos no questionário. O resumo foi gerado pelo serviço Google Docs. O número de respostas obtidas ficou aquém do esperado, dado o vasto número de membros do grupo entrevistado. Apesar de este número rondar os dois milhares e apesar do conteúdo do grupo ser público, ou seja, o número de possíveis entrevistados poder escalar ainda mais, apenas foram obtidas nove respostas. Atribuí-se a fraca adesão ao questionário à falta de um modelo de incentivos para a sua realização, para além da boa vontade dos inquiridos.

Independentemente do tamanho da amostra conseguida é sempre possível tirar algumas conclusões dos resultados obtidos. Na primeira parte do questionário tentou aferir-se o background dos questionados, de forma a poder-se fazer uma correlação entre este e as suas respostas às questões posteriores. Dos resultados observa-se que a generalidade (66%) dos inquiridos tem entre zero e dois anos de experiência. 22% têm mais de 5 anos de experiência, coincidindo este dado com o número de gestores de projecto questionados. Observa-se também que a grande generalidade dos questionados tem bons conhecimentos de tecnologias Web (66 a 77%) e já participaram na construção de uma aplicação através das mesmas (78%). Estes dados são importantes para aferir a qualidade das respostas posteriormente dadas sobre a interface portável desenvolvida. Ainda nesse sentido, questionou-se sobre a familiaridade com os conceitos Wave, sobre o projecto FedOne e sobre o protocolo cliente-servidor Wave. A totalidade dos inquiridos mostra ter pelo menos um conhecimento médio dos conceitos Wave. No entanto os conhecimentos do projecto FedOne encontram-se mais dispersos, tendo 22% apenas conhecimento médio e 33% conhecimento muito alto. Por outro lado, foi possível observar-se que 22% dos inquiridos tem um servidor FedOne e usa-o com colegas ou amigos, 33% tem um servidor FedOne integrado numa plataforma e 11% tem a sua própria implementação de servidor Wave. A conclusão que se faz destes primeiros dados é que a generalidades dos questionados possui não só bons conhecimentos de tecnologias Web e colaborativas como do próprio Wave e do projecto FedOne, sendo portanto a população ideal para responder ao questionário.

Antes de apresentar a interface portável desenvolvida tentou aferir-se sobre a satisfação com o actual protocolo cliente-servidor e cliente do projecto FedOne, importância dada a

portabilidade e eficiência e interesse em ter um protocolo portável e uma interface para desenvolver mais facilmente clientes Wave. Dos resultados observa-se que 55% dos questionados estão pouco ou muito pouco satisfeitos com o *console client* do projecto FedOne, a maior parte estão mediamente satisfeitos com a performance do protocolo cliente-servidor Wave e 66% estão pouco ou muito pouco satisfeitos com a sua portabilidade. Em termos do balanço feito entre portabilidade e performance, a balança pesa ligeiramente mais no sentido da portabilidade, com 33% a darem uma ligeira importância superior a portabilidade, 22% a eficiência e 44% a fazerem um balanço intermédio. Por outro lado, todos os inquiridos mostraram interesse em ter um protocolo cliente-servidor portável mas só 56% mostraram interesse em ter uma interface que facilitasse o desenvolvimento de clientes Web para o Wave. A conclusão tirada destes resultados é que há um interesse real em ter um protocolo cliente-servidor mais portável, sem sacrificar gravemente a sua performance, porém nem todos os inquiridos têm interesse em desenvolver clientes Wave, preferindo usar os desenvolvidos por terceiros. Mais especificamente, os questionados que mostraram menos interesse em desenvolver clientes Wave foram os que ou possuíam menos conhecimentos de tecnologias Web e do projecto FedOne ou só usavam o serviço Google Wave.

Para expor a interface portável desenvolvida foi criada e mostrada a apresentação constante em [77]. Apresentada a interface, tentou-se obter a avaliação da mesma feita pelos questionados. Em primeiro lugar tentou-se aferir as avaliações feitas dos resultados apresentados. Quanto aos resultados de portabilidade, 89% dos inquiridos concordaram que foi obtido um alto ou muito alto nível de portabilidade. Quanto aos resultados de performance, houve uma grande variância entre as avaliações feitas. A conclusão tirada deste resultado é que os resultados apresentados eram pouco explícitos.

De seguida tentou-se aferir a adesão à interface enquanto API de programação. Por um lado questionou-se sobre o nível de abstracção oferecido pela interface, tendo 77% dos inquiridos respondido que o nível era alto ou muito alto. Por outro lado questionou-se quanto fácil parecia ser usar a interface. Aqui os resultados variaram bastante, o que mostra que provavelmente não seria fácil responder à questão apenas com os dados fornecidos. Porém, tentou-se de seguida efectuar questões mais objectivas. A primeira foi quanto tempo os questionados achariam que demoraria a aprender a interface até estarem prontos a usá-la, tendo 44% respondido algumas horas, 22% alguns dias e outros 22% uma semana inteira. A segunda questão foi quanto tempo achavam que demoraria a criar um cliente Web para o Wave com e sem a interface. Com a interface 44% responderam uma semana, 33% duas semanas, 11% três semanas e os restantes 11% mais de um mês. Sem a interface 56% responderam mais de um mês, enquanto apenas 11% responderam uma semana, 22% duas semanas e os restantes 11% três semanas. Comparando as respostas individuais a estas duas questões com os conhecimentos sobre tecnologias Web demonstrados, observa-se que todos os questionados responderam demorar menos tempo com a interface do que sem. Por outro lado, os indivíduos que dizem ter mais conhecimentos Web e sobre o FedOne são os que demorariam menos tempo em ambas as situações e são os que

mostram menos diferença de tempo entre as duas respostas. Os que têm menos conhecimentos nessas áreas são os que demorariam mais tempo e são os que dizem poupar mais tempo com o uso da interface.

Por fim, foi pedido para fazer uma avaliação entre as vantagens de usar a interface e o seu custo de aprendizagem. 44% dos inquiridos responderam que as vantagens venciam em muito o custo e 22% responderam que venciam ligeiramente. Quando questionados sobre se usariam a interface 67% responderam que sim. Analisando as respostas individuais, este resultado conforma com o da pergunta sobre o interesse em ter uma interface que permitisse mais facilmente criar clientes Web para o Wave, à excepção de 11% dos inquiridos que reformularam a sua opinião.

5.4.2. Contribuições Desenvolvidas na Plataforma

1. **Objectivo do teste**

Validar a usabilidade das restantes contribuições realizadas e da plataforma desenvolvida.

2. **Descrição do teste**

O teste consistiu na realização de um questionário sobre as contribuições desenvolvidas na plataforma. O questionário foi dividido em cinco partes. A primeira parte teve o objectivo de aferir o *background* do questionado. A segunda parte foi sobre as funcionalidades colaborativas da plataforma base. As três partes restantes foram sobre as contribuições desenvolvidas, sendo cada parte sobre uma contribuição diferente.

3. **Ambiente de realização do teste**

Uma vez que a plataforma base ainda não entrou em fase de distribuição e, como tal, não possui clientes, as únicas pessoas que a conhecem a fundo são as que trabalham ou já trabalharam no seu desenvolvimento. Assim, a população alvo deste questionário foi esse conjunto de pessoas.

4. **Resultados do teste**

A secção 3 dos anexos mostra um resumo com os resultados obtidos. Pela parte inicial do questionário pode-se inferir que a amostra apresenta um conhecimento consolidado na área de colaboração, tendo 71% dos inquiridos pelos menos 2 anos de experiência na área e a mesma percentagem avaliado o seu conhecimento na mesma como alto, com os restantes 29% a avaliar como médio. Quando questionados sobre as funcionalidades da plataforma base, na segunda parte do questionário, a grande maioria dos inquiridos encontra-as como importantes ou muito importantes. No entanto, todos os questionados acham que a plataforma deveria ter mais

funcionalidades. Aproveitou-se o questionário para obter algum *feedback* sobre que funcionalidades deveriam ser adicionadas à plataforma. Como resposta foram sugeridas funcionalidades de edição de documentos, *DesktopSharing*, *ApplicationSharing* e *widgets* colaborativas.

A terceira parte do questionário foi sobre a integração do Wave na plataforma. Segundo os resultados, 29% dos inquiridos estariam interessados em ter funcionalidades de edição colaborativa de documentos na plataforma e 57% estariam muito interessados. Quanto à escolha do Google Wave para implementar essas funcionalidades, 71% achou que foi uma boa escolha e os restantes 29% achou que foi uma escolha muito boa. Conclui-se portanto que a integração do Wave na plataforma foi, de uma forma geral, bem aceite pelos questionados.

Na quarta parte do questionário questionou-se sobre a aceitação das funcionalidades de *Recording* e *Play* concebidas. Antes de apresentar o trabalho desenvolvido tentou-se aferir sobre o interesse na funcionalidade de reprodução do histórico de uma Wave, funcionalidade que inspirou esta contribuição. 71% dos questionados mostraram interesse ou muito interesse na funcionalidade e 86% mostraram-se interessados em ter algo semelhante na plataforma. Para expor as funcionalidades implementadas foi criada e mostrada a apresentação constante em [78]. Quando questionados sobre a forma como foram concebidas as funcionalidades de *Recording*, 71% achou que foram bem ou muito bem concebidas e 29% achou que foram razoavelmente concebidas. Quanto à completude das funcionalidades, houve exactamente a mesma proporção, ou seja, os inquiridos que encontraram as funcionalidades de *Recording* apenas razoavelmente bem concebidas responderam depois que estas deveriam estar mais completas. Quando inquiridos sobre que funcionalidades faltou conceber, obteve-se respostas como definição de relações temporais entre as *utterances*.

Ainda na quarta parte, inquiriu-se sobre o interesse na concepção das funcionalidades de *Play*, mais especificamente na importação e exportação de documentos. 86% dos inquiridos respondeu que estava interessado ou muito interessado nessas funcionalidades, enquanto os restantes 14% responderam que estavam razoavelmente interessados. Quanto aos formatos contemplados para importação e exportação, também 86% acharam que foram suficientes e 14% insuficientes. Quando questionados sobre que outros formatos estariam interessados obteve-se respostas como Excel e RDF.

Por fim, a quinta parte do questionário foi sobre o modelo de extensibilidade desenvolvido na plataforma. Mais uma vez, antes de apresentar-se o trabalho desenvolvido tentou-se aferir sobre o interesse nas tecnologias que lhe serviram de base. Quando inquiridos sobre as *gadgets* do serviço Google Wave, 86% mostraram-se interessados ou muito interessados nas mesmas e todos os questionados responderam que gostariam de ter o mesmo sistema na plataforma. De seguida foi exposto o trabalho desenvolvido através da apresentação constante em [79]. Quando questionados sobre a forma como foi concebido o modelo de extensibilidade, 71% acharam que foi bem ou muito bem concebido e os restantes 29% acharam que foi apenas razoavelmente bem

concebido. Sobre o tipo de *widgets* que era possível usar, 71% ficaram satisfeitos e 29% ficaram apenas razoavelmente satisfeitos. A conclusão tirada deste resultado é que deveria haver um sistema que permitisse automaticamente adicionar as *gadgets* do Wave à plataforma sem ser preciso converter ficheiros manualmente. Sobre o novo sistema de funcionalidades colaborativas dinâmicas, 86% concordou que havia necessidade da sua concepção e 14% concordou muito com o mesmo. Quanto ao sistema concebido em si, 71% acharam que resolvia o problema eficazmente e 29% acharam que apenas resolvia de forma razoável.

5.5. Conclusões

Este capítulo teve como objectivo validar as contribuições desenvolvidas segundo diferentes critérios. Por um lado houve a necessidade de validar o nível de portabilidade da primeira contribuição desenvolvida, pois o seu objectivo era servir dispositivos heterogéneos. Por outro lado, era necessário validar a performance de todas as contribuições desenvolvidas, de forma a garantir o bom desempenho do trabalho realizado. Por fim foi também validada a usabilidade das diferentes contribuições, através da realização de questionários.

Sobre a primeira contribuição conclui-se que foi atingido, como pretendido, um alto nível de portabilidade. Conclui-se também que, apesar do *tradeoff* que foi necessário fazer entre portabilidade e eficiência, a interface desenvolvida mostra bom desempenho nas suas diferentes operações e face ao desempenho do próprio servidor Wave. Os resultados do questionário de usabilidade mostraram que a generalidade dos inquiridos concorda com esta afirmação e que houve uma reacção positiva ao trabalho desenvolvido e aos seus resultados.

A validação de performance das contribuições desenvolvidas na plataforma mostrou que, nas primeiras avaliações de desempenho realizadas, havia um custo desnecessário resultante do modelo de envio de eventos para as aplicações. Este custo era especialmente evidente na criação e *setup* de sessões. A conclusão inferida é que este era consequência necessária do uso de um tópico de JMS por sessão da plataforma. Concluiu-se portanto que deveria se passar a usar um só tópico para toda a plataforma. Para complementar a análise da solução de JMS, verificou-se a RAM gasta pela pelo servidor JBoss com e sem o *broker* de ActiveMQ integrado. O resultado foi que o ActiveMQ obrigava o JBoss a usar mais 300 MBs de RAM. Como a principal razão de escolha do ActiveMQ foi possibilitar a criação de tópicos dinamicamente, as conclusões retiradas é que deveria também ser abandonado o sistema ActiveMQ e deveria ser usado o sistema de JMS do próprio servidor JBoss.

Depois de realizadas as optimizações descritas no capítulo 4.3.4, que partiram em parte das conclusões do último parágrafo, procedeu-se à reavaliação do desempenho dos diferentes componentes da plataforma. A conclusão retirada dos novos resultados e da comparação entre estes e os resultados originais é que as optimizações introduzidas tiveram um impacto bastante

significativo. Não só foi possível reduzir no tempo de criação e setup de sessões, com a eliminação de um dos componentes que mais tempo gastava, como não se notaram efeitos secundário negativos na utilização das sessões e do sistema de eventos.

Com estes problemas resolvidos o grande *bottleneck* da plataforma, que já era visível nas validações originais e passou a destacar-se ainda mais com a optimização dos outros componentes, era o acesso à base de dados. Este é consequência do seu complexo modelo de dados, que incluí, por exemplo, referências circulares. Este problema foi herdado da plataforma base e, apesar de não ter sido objectivo desta dissertação resolvê-lo, tentou-se minimizar os custos adicionais resultantes das contribuições desenvolvidas. Porém, a conclusão tirada dos resultados é que deve ser realizado um trabalho de reestruturação do modelo de dados.

Para além da base de dados, outros *bottlenecks* de menores dimensões eram os servidores de recursos. Especialmente no caso do servidor Wookie, a conclusão retirada é que deverão ser analisados no futuro formas de optimizar a comunicação com os mesmos ou deverão mesmo ser procurados servidores com as mesmas funcionalidades mas mais eficientes.

Por fim, a validação da usabilidade das contribuições desenvolvidas na plataforma permitiu retirar informações importantes sobre a completude e eficácia das mesmas. Não obstante de alguns pormenores que poderiam estar mais completos ou ter sido implementados de forma diferente, houve uma reacção geral positiva ao trabalho realizado e à necessidade do mesmo.

6. Conclusões

O trabalho realizado nesta dissertação relata o objectivo de modelar e desenvolver uma plataforma extensível que agrupasse diversos recursos colaborativos sob uma interface unificada e que permitisse facilitar e fomentar a construção de aplicações colaborativas complexas a executar em dispositivos heterogéneos.

A partir da análise de um conjunto de trabalhos na área de colaboração, nomeadamente ao nível do estudo de serviços, plataformas e tecnologias colaborativas mais usados hoje em dia, surgiu a motivação para desenvolver algo que pudesse oferecer novas soluções. Mais especificamente, da análise crítica dos conceitos e tecnologias apresentados pelo Google Wave, assim como dos seus protocolos comunicacionais, técnicas para controlo de concorrência e modelos de extensibilidade, surgiu a inspiração para desenvolver uma série de contribuições que levariam à realização do referido objectivo desta dissertação.

A Plataforma Unificada de Colaboração (PUC), que está a ser desenvolvida na PT Inovação, oferece alguns dos requisitos identificados anteriormente. Mais concretamente, a plataforma une funcionalidades de conferências de áudio e vídeo, *Slideshow*, *Whiteboarding*, partilha de ficheiros, conferências SIP, entre outras, sob uma única interface colaborativa genérica. No entanto pretendia-se uma plataforma com mais funcionalidades colaborativas, mais extensível e com mais suporte a heterogeneidade.

Assim partiu-se para o desenvolvimento de uma plataforma mais avançada, a partir da plataforma PUC e através da integração na mesma das funcionalidades e conceitos Wave mais marcantes.

De seguida apresenta-se os objectivos enunciados no capítulo da introdução, confrontando-os com o que foi atingido no trabalho efectuado nesta dissertação.

6.1. Objectivos Revistos

Acredita-se que todos os objectivos propostos foram atingidos. A plataforma desenvolvida simplifica e facilita a construção de aplicações colaborativas complexas, como demonstrado em [41] e pela figura 1 presente no capítulo 1.3. Pela extensa validação feita no capítulo 5, conclui-se também que as contribuições desenvolvidas não só conformem com os requisitos especificados como o fazem com boa performance e demonstram bons resultados de usabilidade.

Tendo em contas as contribuições propostas relativamente ao trabalho a desenvolver, discute-se de seguida as que foram objectivamente alcançadas:

Unidade Foi reforçada a interface unificada genérica da plataforma PUC através da inclusão na mesma de funcionalidades de edição colaborativa de documentos e através da integração do Wave como novo servidor de recursos.

Heterogeneidade A interface portável concebida para o Wave reforça o suporte a dispositivos heterogéneos na plataforma desenvolvida. O nível de portabilidade alcançado foi validado no capítulo 5.1.

Extensibilidade O modelo de extensibilidade desenvolvido na plataforma permite adicionar-lhe um número de funcionalidades colaborativas praticamente ilimitado, como comprovado pelo vasto número de *gadgets* do Google Wave existentes na Web e que podem, através do trabalho desenvolvido, ser incorporadas nas sessões da plataforma. Este modelo, aliado ao sistema dinâmico de carregamento de funcionalidades colaborativas concebido, permitiu introduzir na plataforma PUC um nível de extensibilidade sem precedentes.

Fiabilidade Defende-se que a plataforma desenvolvida é fiável. Esta fiabilidade provém, por um lado, do seu modelo de persistência, herdado da plataforma PUC e reforçado nesta dissertação e nas contribuições desenvolvidas. Por outro, a fiabilidade resulta também do sistema concebido para entrega assíncrona de eventos às aplicações.

Usabilidade Através da adição de funcionalidades de edição colaborativa de documentos à plataforma e de um modelo de extensibilidade baseado em *widgets* colaborativas, através da adição de funcionalidades de importação e exportação de documentos no Wave e através do desenvolvimento de funcionalidades de *Recording* e *Play* foi alcançado, com esta dissertação, um nível de usabilidade que superou as expectativas iniciais. Este nível foi validado no capítulo 5.3, com dois questionários de usabilidade.

6.2. Trabalho Futuro

Nesta secção irão ser numeradas um conjunto de funcionalidades e actividades identificadas como importantes e prioritárias num futuro próximo. Estas são essencialmente resultantes das conclusões tiradas dos processos de validação efectuados no capítulo 5.

As alterações ou futuras adições sugeridas à plataforma são:

- Implementar a entidade *SessionDocument* como uma classe embebida na entidade *Session* e validar os resultados obtidos, em termos de performance de acessos à base de dados.

- Definição de relações temporais entre *utterances*, nas funcionalidades de *Recording* e *Play*. Com esta nova funcionalidade passará a ser possível definir uma sequência temporal

precisa entre todos os recursos colaborativos usados na duração de um excerto de sessão, aumentando ainda mais a usabilidade desta contribuição.

- Optimização do modelo de dados da plataforma e de acessos a base de dados, que é o actual *bottleneck* nos testes de performance realizados.

- Alargamento de escala e tipo de testes de usabilidade.

- Revisão da plataforma e alargamento de testes de desempenho, para desenvolvimento de um protótipo mais abrangente de teste de integração/pré-produção.

Bibliografia

[1] http://www.ptinovacao.pt/ (consultado pela última vez a 28/07/2010).

[2] Baecker, Ronald M. *Readings in Groupware Computer-Supported Cooperative Work*. Morgan Kaufmann Publishers, San Francisco, California, 1993.

[3] A. P. Correia, "A Framework for Collaborative Applications", Dissertação de Mestrado em Engenharia Informática, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa (work in progress).

[4] http://code.google.com/p/wave-protocol/ (consultado pela última vez a 28/07/2010).

[5] http://wave.google.com/about.html (consultado pela última vez a 28/07/2010).

[6] http://code.google.com/intl/pt-PT/apis/protocolbuffers/ (consultado pela última vez a 28/07/2010).

[7] M. Weiser. The computer for the 21st century. SIGMOBILE Mob. Comput. Commun. Rev., 3(3):3–11, 1999.

[8] http://code.google.com/intl/pt-PT/apis/wave/ (consultado pela última vez a 28/07/2010).

[9] C. A. Elfis and S. J. Gibbs: "Concurrency control in groupware systems", In Proc. of ACM SIGMOD Conference on Management of Data, pp.399-407, 1989.

[10] C. Sun, Y.Yang, Y. Zhang, ad D. Chen: "A consistency model and supporting schemes for real-time cooperative editing systems", In Proc. of The 19th Australasian Computer Science Conference, pp. 582-591, Melbourne, Jan. 1996.

[11] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenbauser:"An integrating, transformation-oriented approach to concurrency control and undo in group editors", In Proc. of ACM Conference on Computer Supported Cooperative Work, pp 288-297, Nov. 1996.

[12] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen: "Achieving convergence, causatity-preservation, and intention-preservation in real-time cooperative editing systems", ACM Transactions on Computer-human Interaction, 5(1), March 1998, pp.63-108.

[13] D. Nichols, P. Curtis, M. Dixon, and J. Lamping “High-latency, low-badwidth windowing in the Jupiter collaboration system: In Proc. of ACM Symposium on User Interface Software and Technologies, pp. 111-120, Nov. 1995.

[14] C. Sun; C. Ellis. "Operational transformation in real-time group editors: issues, algorithms, and achievements". Proceedings of the 1998 ACM conference on Computer supported cooperative work. ACM Press New York, NY, USA. pp. 59-68, 1998.

[15] D. Wang, A. Mah, "Google Wave Operational Transformation", Google Wave Federation Protocol White Papers, http://www.waveprotocol.org/whitepapers/operational-transform (consultado pela última vez a 05/02/2010).

[16] http://xmpp.org/ (consultado pela última vez a 28/07/2010).

[17] http://docs.google.com/support (consultado pela última vez a 28/07/2010).

[18] S. Dekeyser, R. Watson, "Extending Google Docs to Collaborate on Research Papers", Technical Report, University of Southern Queensland, Australia, 2006.

[19] P. Saint-Andre, Ed., “Extensible Messaging and Presence Protocol (XMPP): Core,” RFC 3920, October 2004.

[20] A. Baxter, J. Bekmann, D. Berlin, S. Lassen, S. Thorogood, "Google Wave Federation Protocol Over XMPP", Google Wave Federation Protocol Draft Protocol Specs, July 2009, http://www.waveprotocol.org/draft-protocol-specs/draft-protocol-spec (consultado pela última vez a 28/07/2010).

[21] http://www.ietf.org/dyn/wg/charter/simple-charter.html (consultado pela última vez a 28/07/2010).

[22] A. Houri, E. Aoki, S. Parameswar, T. Rang, V. Singh, H. Schulzrinne, “Presence Interdomain Scaling Analysis for SIP/SIMPLE”, Internet Draft, IETF SIMPLE WG, August 2009 (work in progress).

[23] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler, “SIP: Session Initiation Protocol”, RFC 3261, June 2002.

[24] P. Millard, P. Saint-Andre, R. Meijer, “XEP-0060: Publish-Subscribe”, Standards Track Draft, XMPP Standards Foundation, December 2009 (work in progress).

[25] http://code.google.com/p/openmeetings/ (consultado pela última vez a 28/07/2010)

[26] http://os_ash.org/red5 (consultado pela última vez a 28/07/2010)

[27] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, W. Cai (2006). "Transparent Adaptation of Single-User Applications for Multi-User Real-Time Collaboration". ACM Transactions on Computer-Human Interaction, Vol. 13, No. 4, December 2006, Pages 531–582.

[28] C. SUN 2002. Undo as concurrent inverse in group editors. *ACM Trans. Comput.-Human Interact.9*, 4, 309–361.

[29] http://WebEx.com/what-is-WebEx/index.html (consultado pela última vez a 28/07/2010)

[30] "ITU-T Recommendation T.120, Data protocols for multimedia conferencing", Telecommunication Standardization Sector of International Telecommunication Union, 13 January 2007.

[31] "A Primer on the T.120 Standard", DataBeam Corporation, Lexington Kentucky, 1997.

[32] http://www.novell.com/products/pulse/ (consultado pela última vez a 28/07/2010).

[33] http://pygowave.net/ (consultado pela última vez a 28/07/2010).

[34] P. Saint-Andre, "XEP-0114: Jabber Component Protocol", Historical Extension, XMPP Standards Foundation, March 2005.

[35] http://www.igniterealtime.org/projects/openfire/index.jsp (consultado pela última vez a 28/07/2010).

[36] A. Russel, G. Wilkins, D. Davis and M. Nesbit: "The Bayeux Specification", The Dojo Foundation, 2007, http://svn.cometd.org/trunk/bayeux/bayeux.html

[37] I. Paterson, D. Smith, P. Saint-Andre and J. Moffitt: "XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH)", Standards Track Draft, XMPP Standards Foundation, November 2009

[38] I. Hickson: "The WebSocket Protocol", Internet-Draft, IETF Network WG, 6 May 2010 (work in progress)

[39] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee: "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, IETF Networking Group, June 1999

[40] I. Hickson: "The WebSocket API", Working Draft, W3C, 22 December 2009 (work in progress)

[41] J. Sousa, "Mobi-Collab: Collaborative Applications for Mobile Devices", Dissertação de Mestrado em Engenharia de Redes de Comunicações, Instituto Superior Técnico (work in progress)

[42] M. Cáceres, “Widget Packaging and Configuration”, W3C Candidate Recommendation, World Wide Web Consortium, 01 December 2009

[43] http://code.google.com/intl/pt-PT/apis/opensocial/ (consultado pela última vez a 28/07/2010).

[44] http://code.google.com/intl/pt-PT/apis/wave/extensions/gadgets/guide.html (consultado pela última vez a 28/07/2010).

[45] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E.Maler, F. Yergeaus, “Extensible Markup Language (XML) 1.0 (Fifth Edition)”, W3C Recommendation, World Wide Web Consortium, 26 November 2008.

[46] D. Raggett, A. Le Hors, I. Jacobs, “HTML 4.01 Specification”, W3C Recommendation, World Wide Web Consortium, 24 December 1999.

[47] "Information Processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML)", ISO 8879:1986, International Organization for Standardization.

[48] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne, “Document Object Model (DOM) Level 3 Core Specification”, W3C Recommendation, World Wide Web Consortium, 07 April 2004.

[49] http://www.saxproject.org/ (consultado pela última vez a 28/07/2010).

[50] http://xerces.apache.org/ (consultado pela última vez a 28/07/2010).

[51] http://dev.plutext.org/trac/docx4j (consultado pela última vez a 28/07/2010).

[52] https://xhtmlrenderer.dev.java.net/ (consultado pela última vez a 28/07/2010).

[53] http://getwookie.org/Welcome.html (consultado pela última vez a 28/07/2010).

[54] http://shindig.apache.org/ (consultado pela última vez a 28/07/2010).

[55] “Java Remote Method Invocation Specification”, Sun Microsystems, 2004

[56] http://github.com/danopia/ruby-on-sails (consultado pela última vez a 28/07/2010).

[57] “Oracle WebLogic Server Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server, 10*g* Release 3 (10.3)”, Oracle, July 2008

[58] “IBM WebSphere Application Server Feature Pack for Web 2.0 Web messaging servide”, IBM Software Group, 17 December 2007, http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/topic/com.ibm.iea.wasfpweb20/w

asfpweb20/6.0/Messaging/was_web20fp_Web-Messaging-Service.pdf (consultado pela última vez a 28/07/2010).

[59] http://cometd.org/ (consultado pela última vez a 28/07/2010).

[60] J. Gregorio, A. North, "Google Wave Conversation Model", Google Inc., October 2009, http://wave-protocol.googlecode.com/hg/spec/conversation/convspec.html (consultado pela última vez a 28/07/2010).

[61] http://cometd.org/documentation/cometd-java (consultado pela última vez a 28/07/2010).

[62] http://java.sun.com/products/servlet/ (consultado pela última vez a 28/07/2010).

[63] http://jetty.codehaus.org/jetty/ (consultado pela última vez a 28/07/2010).

[64] B. Shannon, "Java Platform, Enterprise Edition 5 (Java EE 5) Specification", Sun Microsystems, 8 May, 2006

[65]http://docs.jboss.org/jbossas/docs/Server_Configuration_Guide/beta500/html/index.html (consultado pela última vez a 28/07/2010).

[66] L. DeMichiel, M. Keith, "JSR 220: Enterprise JavaBeans v.3.0, EJB Core Contracts and Requirements", Sun Microsystems, 8 May, 2006

[67]http://www.jboss.org/ejb3/docs/reference/build/reference/en/html/jboss_extensions.html (consultado pela última vez a 28/07/2010).

[68] M. Hapner, R. Burridge, R. Sharma, J. Fialli, K. Stout, "Java Message Service Specification", Sun Microsystems, April 12, 2002

[69] http://activemq.apache.org/ (consultado pela última vez a 28/07/2010).

[70] http://www.w3schools.com/tags/default.asp (consultado pela última vez a 28/07/2010).

[71] http://docs.jboss.org/jbossas/jboss4guide/r1/html/ch10.html (consultado pela última vez a 28/07/2010).

[72] http://www.postgresql.org/ (consultado pela última vez a 28/07/2010).

[73] http://code.google.com/intl/pt-PT/webtoolkit/overview.html (consultado pela última vez a 28/07/2010).

[74] http://www.eclipse.org/tptp/ (consultado pela última vez a 28/07/2010).

[75] http://jboss.org/tools (consultado pela última vez a 28/07/2010).

[76]http://download.oracle.com/docs/cd/E17477_01/javaee/5/api/javax/persistence/Embeddable.html (consultado pela última vez a 28/07/2010).

[77] http://docs.google.com/present/view?id=d7bzfkx_42bhwh96v&interval=60 (consultado pela última vez a 28/07/2010).

[78] http://docs.google.com/present/view?id=d7bzfkx_29gc4cs6hk&interval=60 (consultado pela última vez a 28/07/2010).

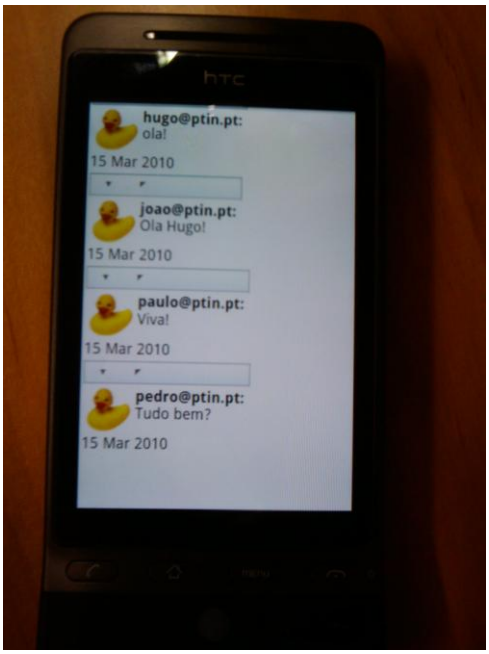
[79] http://docs.google.com/present/view?id=d7bzfkx_35sr45hxgm&interval=60 (consultado pela última vez a 28/07/2010).

[80] http://groups.google.com/group/wave-protocol/ (consultado pela última vez a 28/07/2010).

[81] http://docs.jboss.org/jbossmessaging/docs/guide-1.0.1.SP5/html_single/index.html (consultado pela última vez a 28/07/2010)

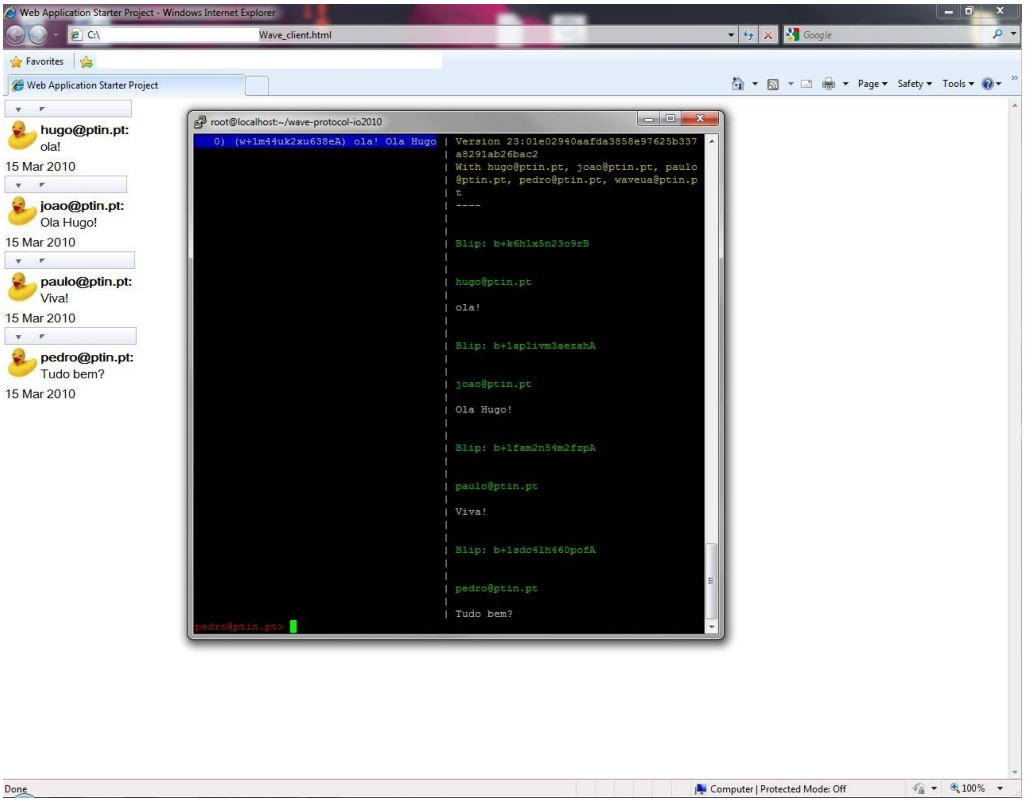
Documentação Anexa

Secção 1 – Testes de portabilidade efectuados à interface portável para o Wave



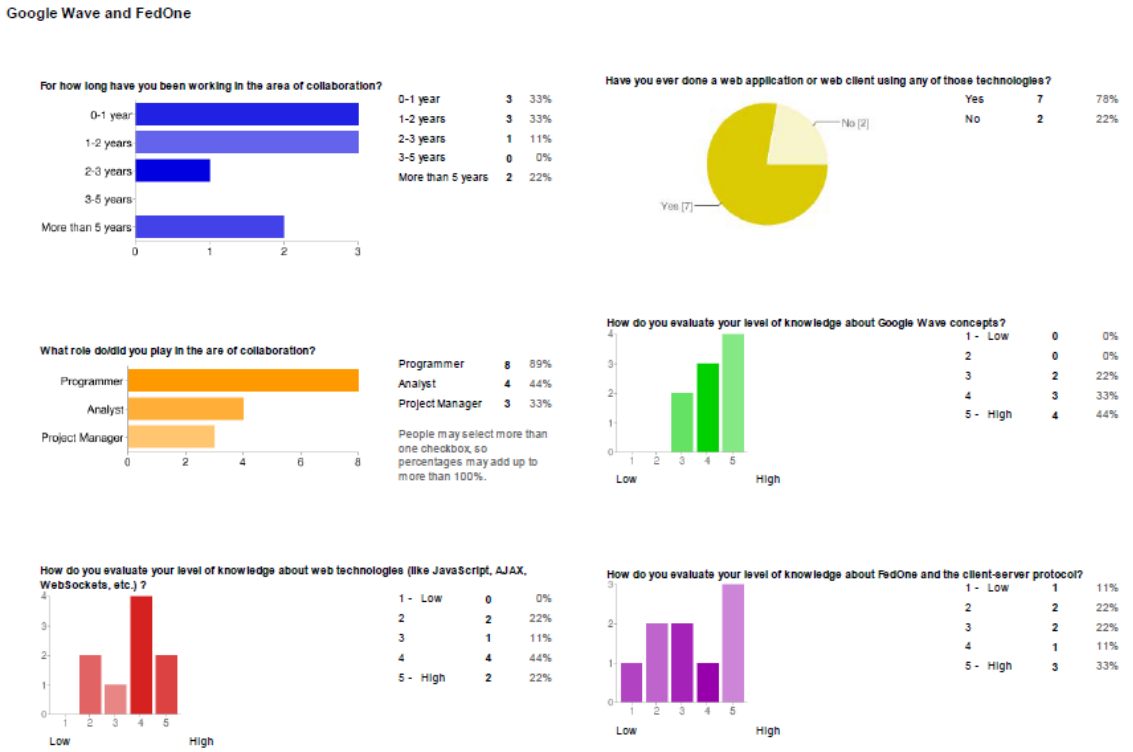
Anexo 1 – HTC Hero com Android 2.1 ligado à interface portável desenvolvida

Anexo 2 – Ipod Touch com cliente Wave ligado à interface portável desenvolvida



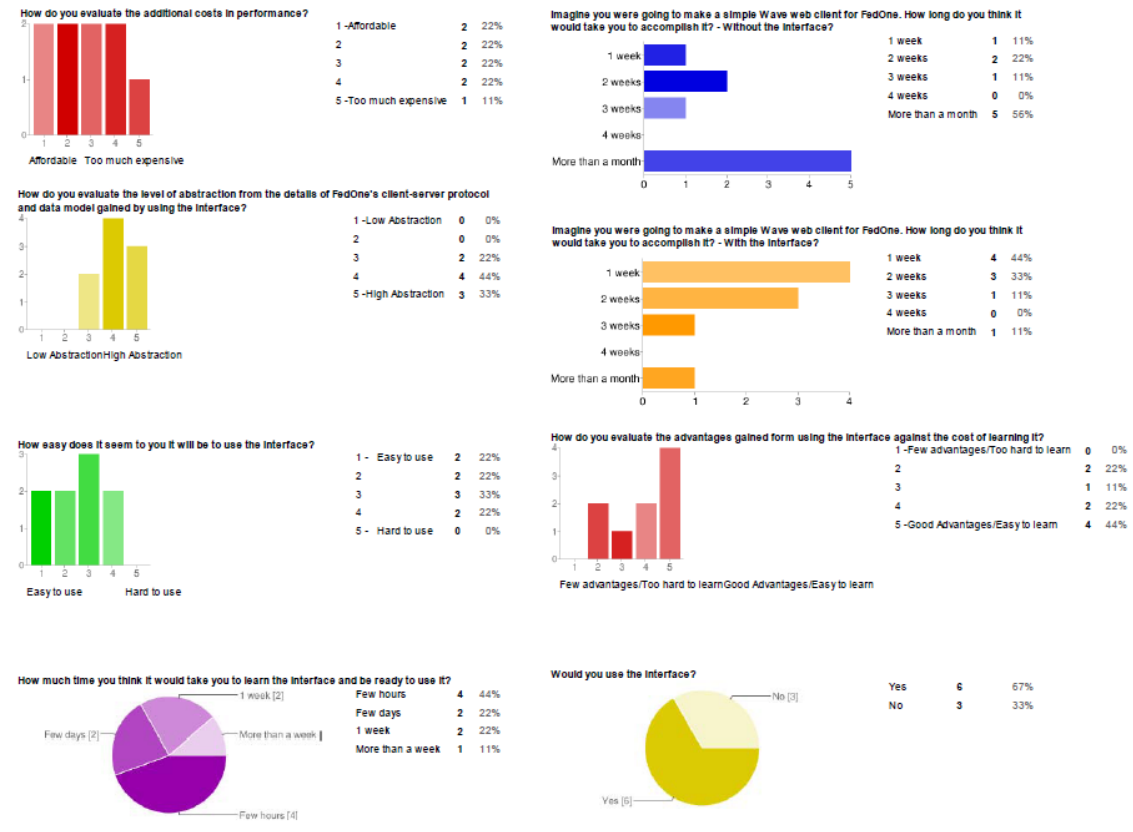
Anexo 3 – Internet Explorer 8 com cliente Wave ligado à interface portável desenvolvida, juntamente com *console client* do projecto FedOne, ambos ligados à mesma Wave

Secção 2 – Resultados do teste de usabilidade da interface portável para o Wave



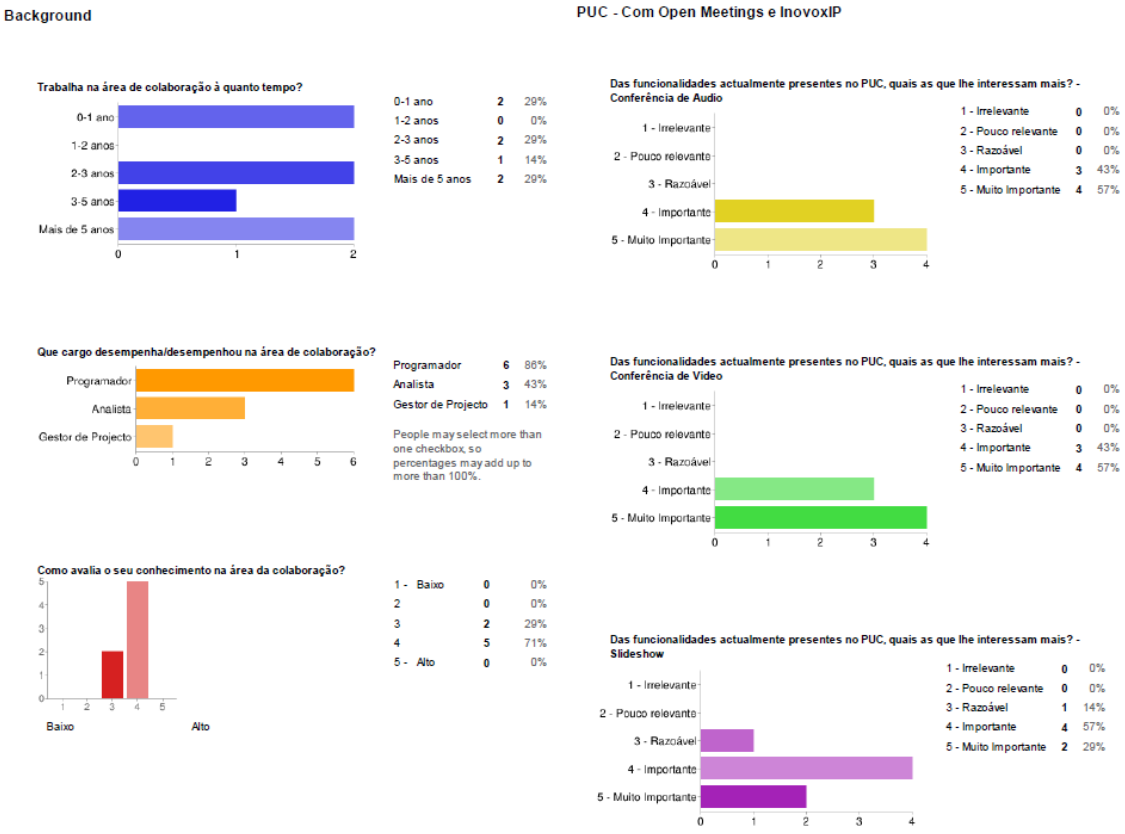
Anexo 4 – Resultados do teste de usabilidade da interface portável para o Wave, parte 1

Anexo 5 – Resultados do teste de usabilidade da interface portável para o Wave, parte 2

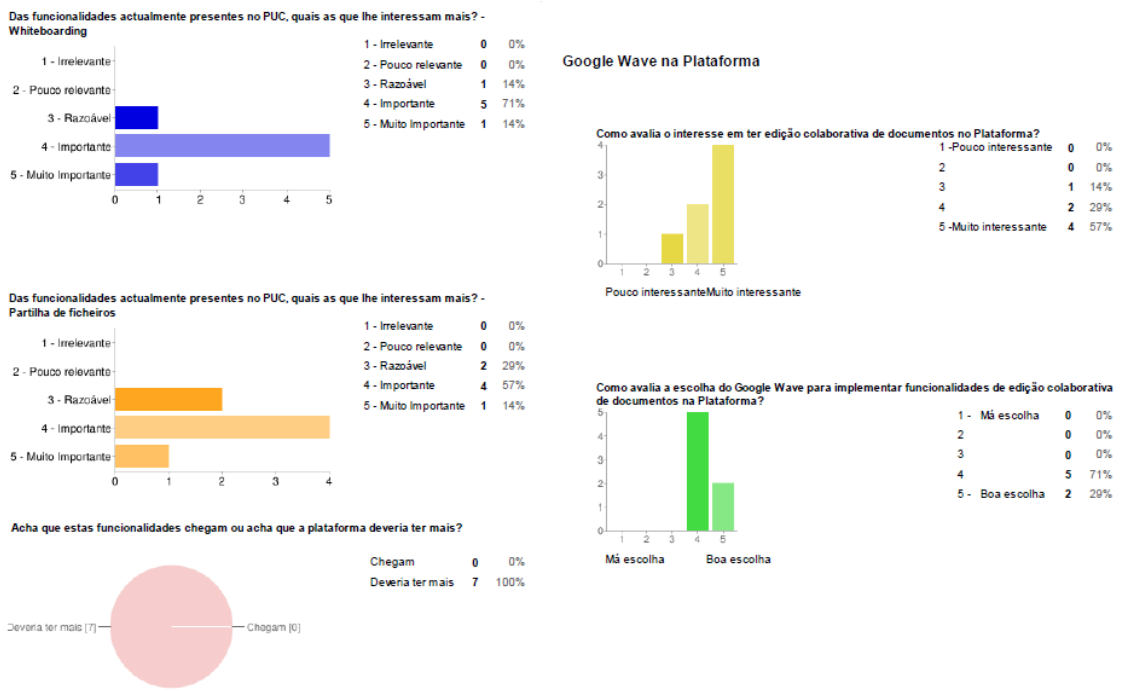


Anexo 6 – Resultados do teste de usabilidade da interface portável para o Wave, parte 3

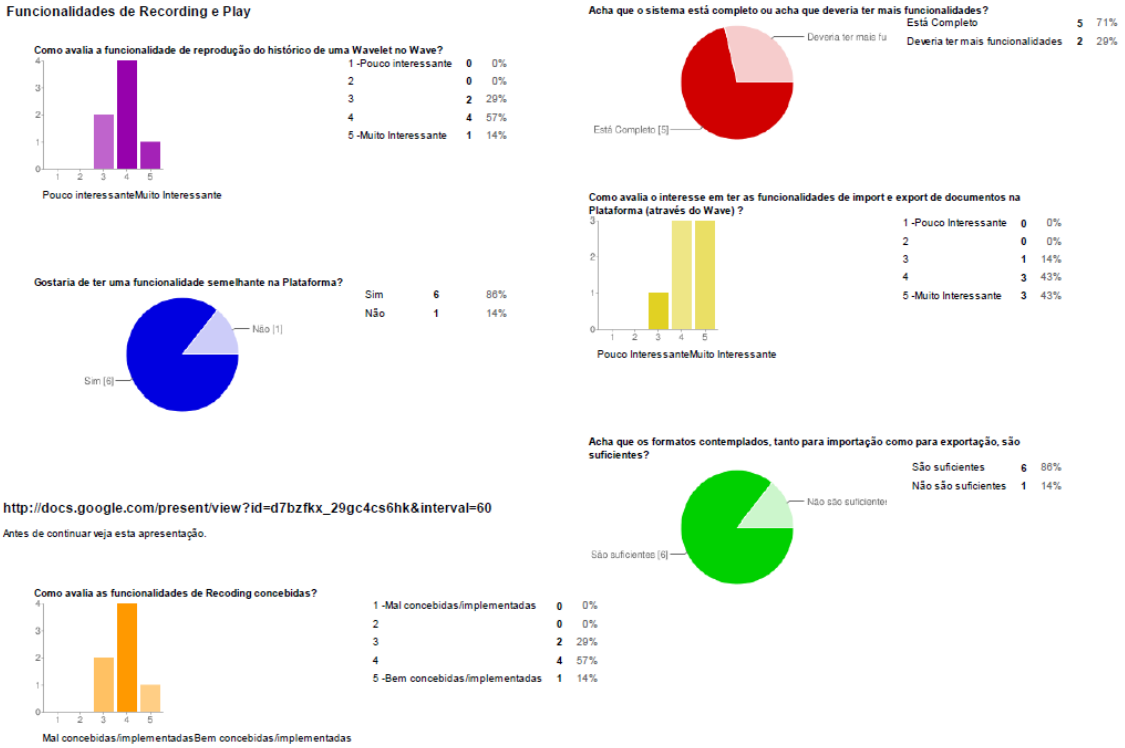
Secção 3 – Resultados do teste de usabilidade sobre as contribuições desenvolvidas na plataforma



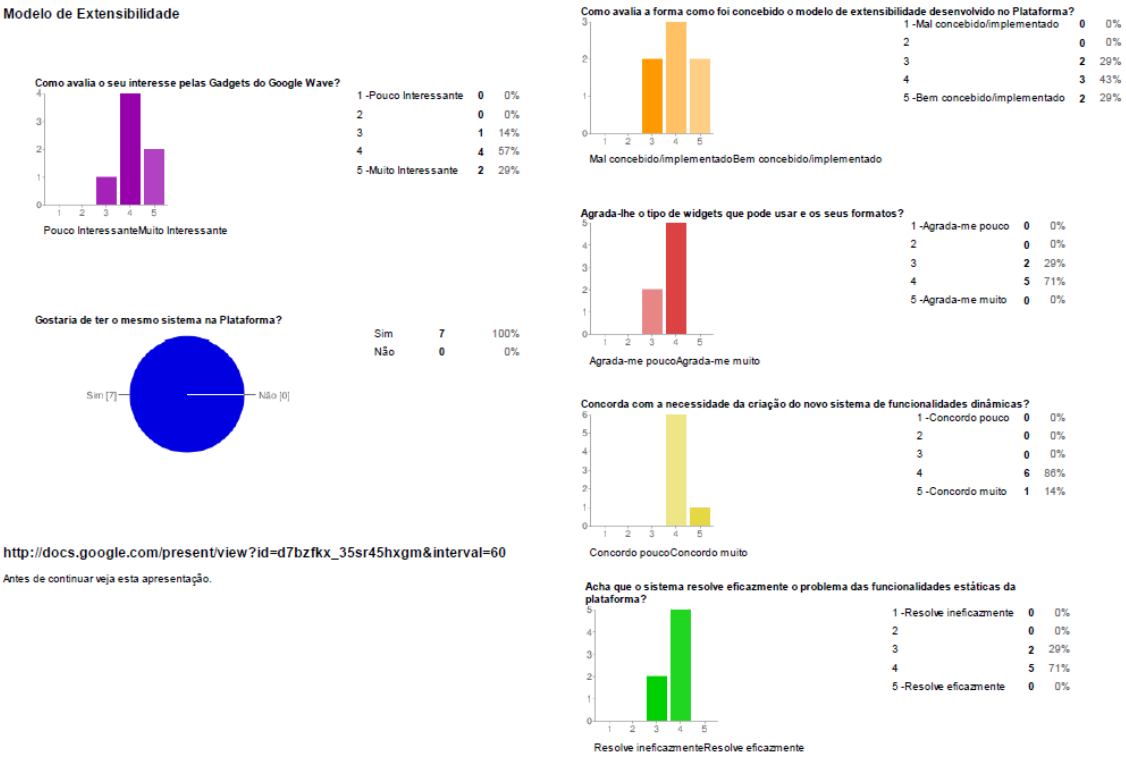
Anexo 7 – Resultados Resultados do teste de usabilidade sobre as controbuições desenvolvidas na plataforma, parte 1



Anexo 8 – Resultados Resultados do teste de usabilidade sobre as controbuições desenvolvidas na plataforma, parte 2



Anexo 9 – Resultados Resultados do teste de usabilidade sobre as controbuições desenvolvidas na plataforma, parte 3



Anexo 10 – Resultados Resultados do teste de usabilidade sobre as controbuições desenvolvidas na plataforma, parte 4